版权注意事项：
1、书籍版权归作者和出版社所有
2、本PDF仅限用于个人获取知识，进行私底下的知识交流
3、PDF获得者不得在互联网上以任何目的进行传播
4、如觉得书籍内容很赞，请购买正版实体书，支持作者
5、请于下载PDF后24小时内删除本PDF。

Broadview
www.broadview.com.cn

CDA数据分析师系列丛书



Spark大数据分析
技术与实战

经管之家 主编　董轶群 曹正凤 赵仁乾 王安 编著

Spark大数据分析技术全覆盖：Spark GraphX、Spark SQL、Spark Streaming和Spark MLlib；
零基础学习Spark大数据分析技术：有数据、有案例、有分析。

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn



董轶群，吉林大学计算机科学与技术学院博士毕业。曾在吉林大学“符号计算与知识工程”教育部重点实验室从事空间关系建模研究，参与了多个国家自然科学基金重点项目与面上项目的申报与研究工作，并在项目中主要负责空间方向关系建模、空间拓扑关系建模的研究工作。

目前作为经管之家（原人大经济论坛）大数据讲师，主讲Spark、Hbase、Scala等大数据核心课程，并从事大数据相关的理论与应用研究工作。重点关注海量数据背景下空间关系建模与智能交通的结合研究，并在国内期刊和国际会议上发表了一系列相关理论的研究成果。



曹正凤，统计学博士，经管之家（原人大经济论坛）大数据中心总工程师，经管之家CDA大数据分析师培训负责人，北京博宇通达科技有限公司技术总监。致力于大数据分析前沿领域研究，主持首发集团智慧交通大数据中心建设项目，基于大数据平台的互联网金融风险监控系统项目，参与国家社科基金项目《基于大数据整合的空气质量测度方法研究》。



赵仁乾，北京邮电大学管理科学与工程硕士，现就职于北京电信规划设计院任高级经济师，从事移动、联通集团及各省分公司市场、业务、财务规划，经济评价及运营咨询。重点研究方向包括离网用户挖掘、市场细分与精准营销、移动网络价值区域分析、潜在价值客户挖掘等。



王安，布本智能首席数据官，北京大学光华管理学院MBA，北京大学商务智能中心专家组成员。专注数据化决策，互联网金融风险管理与精准营销。在数据决策领域拥有十多年的实践经验，曾服务多家大中型银行、保险公司及互联网金融公司。同时也积极参与数据决策教育领域，为北京大学、人民大学、北京航空航天大学、北京理工大学等院校机构提供相关课程和数据教育辅导。

CDA数据分析师系列丛书

Spark大数据分析技术与实战

经管之家 主编 董轶群 曹正凤 赵仁乾 王安 编著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

Spark 作为下一代大数据处理引擎，经过短短几年的飞跃式发展，正在以燎原之势席卷业界，现已成为大数据产业中的一股中坚力量。

本书着重讲解了 Spark 内核、Spark GraphX、Spark SQL、Spark Streaming 和 Spark MLlib 的核心概念与理论框架，并提供了相应的示例与解析。

全书共分为 8 章，其中前 4 章介绍 Spark 内核，主要包括 Spark 简介、集群部署、工作原理、核心概念与操作等；后 4 章分别介绍 Spark 内核的核心组件，每章系统地介绍 Spark 的一个组件，并附以相应的案例分析。

本书适合作为高等院校计算机相关专业的研究生学习参考资料，也适合大数据技术初学者阅读，还适合所有愿意对大数据技术有所了解并想要将大数据技术应用于本职工作的读者阅读。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

Spark 大数据分析技术与实战 / 经管之家主编；董轶群等编著. —北京：电子工业出版社，2017.7
（CDA 数据分析师系列丛书）
ISBN 978-7-121-31903-7

Ⅰ. ①S… Ⅱ. ①经… ②董… Ⅲ. ①数据处理软件－技术培训－教材 Ⅳ. ①TP274

中国版本图书馆 CIP 数据核字(2017)第 133619 号

策划编辑：张慧敏
责任编辑：徐津平
特约编辑：顾慧芳
印　　刷：北京中新伟业印刷有限公司
装　　订：北京中新伟业印刷有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编：100036
开　　本：787×980　1/16　印张：14.5　字数：330 千字
版　　次：2017 年 7 月第 1 版
印　　次：2017 年 7 月第 1 次印刷
定　　价：59.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

前　言

随着电子信息、物联网等产业的高速发展，智能手机、平板电脑、可穿戴设备与物联网设备已经渗入到现代化生产与生活的各个方面，每时每刻产生着大量的数据，当今社会已经进入数据爆炸的时代。各领域中的相关数据不仅量大，而且种类繁多、变化速度快、价值密度低。这些日益凸显的大数据特征在全球范围内掀起了一场全新的思维、技术与商业变革，无论是产业界还是学术界都在持续加大在大数据技术和相关领域中的投入。“中国制造 2025”战略规划和“互联网+”概念的提出再次为国内大数据技术的发展注入了强劲的动力，大数据技术已被提升到了前所未有的高度，预示了其未来广阔的发展空间与应用前景。

在大数据背景下，各领域对数据相关服务的需求不断提升，迫切需要一种高效通用的大数据处理引擎。相对于第一代大数据生态系统 Hadoop 中的 MapReduce，Spark 是一种基于内存的、分布式的大数据处理引擎，其计算速度更快，更加适合处理具有较多迭代次数的问题；Spark 中还提供了丰富的 API，使其具有极强的易用性；与此同时，Spark 实现了“一栈式”的大数据解决方案，即在 Spark 内核基础上提出了 Spark GraphX、Spark Streaming、Spark MLlib、Spark SQL 等组件，使其不仅能够对海量数据进行批处理，同时还具备流式计算、海量数据交互式查询等功能，可以满足包括教育、电信、医疗、金融、电商、政府、智慧城市和安全等诸多领域中的大数据应用需求。

Spark 作为下一代大数据处理引擎，经过短短几年的飞跃式发展，正在以燎原之势席卷业界，现已成为大数据产业中的一股中坚力量。

本书主要针对大数据技术初学者，着重讲解了 Spark 内核、Spark GraphX、Spark SQL、Spark Streaming 和 Spark MLlib 的核心概念与理论框架，并提供了相应的示例与解析，以便读者能够尽快了解 Spark。

全书共分为 8 章，其中前 4 章介绍 Spark 内核，主要包括 Spark 简介、集群部署、工作原理、核心概念与操作等；后 4 章分别介绍 Spark 内核的核心组件，每章系统地介绍 Spark 的一个组件，并附以相应的案例分析。

- 第 1 章：Spark 导论。概述 Spark 的发展背景与起源，对比 MapReduce 介绍了 Spark 的特征、原理与应用场景等。
- 第 2 章：Spark 集群部署。该章详细介绍了 Ubuntu 下 Spark 集群的部署过程与注意事项，首先利用 VMware Workstation 搭建 Hadoop 分布式集群；然后在集群中安装 Scala；最后搭建 Standalone 模式的 Spark 集群。

- 第 3 章：RDD 编程。该章对 Spark 中的弹性分布式数据集（Resilient Distributed Dataset — RDD）这一核心概念进行了详细介绍，重点讲解了与之相关的定义、特征及其操作，并附以相应的示例与解析。
- 第 4 章：Spark 调度管理与应用程序开发。该章阐述了 Spark 底层的工作机制，介绍了 Spark 应用程序从产生作业到最终计算任务的整个流程；基于 IntelliJ IDEA 讲解了 Spark 应用程序的开发过程，并介绍了如何在本地与集群模式下提交运行 Spark 应用程序。
- 第 5 章：GraphX。该章介绍了 GraphX 的基本原理，着重讲解了 GraphX 中弹性分布式属性图的定义、表示模型、存储方式以及其上的丰富操作；以经典的 PageRank 与三角形计数等图计算算法为例，讲解了 GraphX 中相关接口的使用方法。
- 第 6 章：Spark SQL。该章包含了 Spark SQL 概述、SQL 语句的处理流程、DataFrame 数据模型的概念与相关操作等；并将 Spark SQL 与 Hive 相结合，给出了一个学生信息管理系统的设计与实现。
- 第 7 章：Spark Streaming。该章介绍了 Spark Streaming 的发展与应用场景以及批处理时间间隔、窗口间隔、滑动间隔等核心概念；着重讲解了 DStream 数据模型的概念与相关操作；针对不同应用场景下的流式计算需求，给出了有状态与无状态模式下的 Spark Streaming 应用案例与解析。
- 第 8 章：Spark MLlib。该章介绍了 Spark MLlib 中向量、LabeledPoint、矩阵等核心数据类型的定义与使用；详细介绍了机器学习中分类、回归、聚类、协同过滤等经典算法的 Spark 实现与应用，并附以相应的案例与解析。

由于时间短，加之笔者水平有限，书中难免有疏漏之处，敬请读者朋友批评指正。

编　者

2017 年 5 月

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

- **提交勘误**：您对书中内容的修改意见可在 提交勘误 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **交流互动**：在页面下方 读者评论 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：http://www.broadview.com.cn/31903



目　录

第 1 章　Spark 导论 1

1.1　Spark 的发展 2
1.2　什么是 Spark 3
1.3　Spark 主要特征 3
1.3.1　快速 3
1.3.2　简洁易用 5
1.3.3　通用 6
1.3.4　多种运行模式 8

第 2 章　Spark 集群部署 9

2.1　运行环境说明 9
2.1.1　软硬件环境 9
2.1.2　集群网络环境 10
2.2　安装 VMware Workstation 11 10
2.3　安装 CentOS 6 16
2.4　安装 Hadoop 21
2.4.1　克隆并启动虚拟机 21
2.4.2　网络基本配置 24
2.4.3　安装 JDK 27
2.4.4　免密钥登录配置 28
2.4.5　Hadoop 配置 29
2.4.6　配置从节点 33
2.4.7　配置系统文件 33
2.4.8　启动 Hadoop 集群 33
2.5　安装 Scala 35
2.6　安装 Spark 36
2.6.1　下载并解压 Spark 安装包 36

2.6.2 配置 Spark-env.sh 37
2.6.3 配置 Spark-defaults.conf 37
2.6.4 配置 Slaves 38
2.6.5 配置环境变量 38
2.6.6 发送至 Slave1、Slave2 39
2.7 启动 Spark 39

第 3 章 RDD 编程 42

3.1 RDD 定义 42
3.2 RDD 的特性 43
3.2.1 分区 43
3.2.2 依赖 44
3.2.3 计算 45
3.2.4 分区函数 45
3.2.5 优先位置 46
3.3 创建操作 46
3.3.1 基于集合的创建操作 47
3.3.2 基于外部存储的创建操作 47
3.4 常见执行操作 49
3.5 常见转换操作 49
3.5.1 一元转换操作 50
3.5.2 二元转换操作 53
3.6 持久化操作 56
3.7 存储操作 58

第 4 章 Spark 调度管理与应用程序开发 59

4.1 Spark 调度管理基本概念 59
4.2 作业调度流程 60
4.2.1 作业的生成与提交 61
4.2.2 阶段的划分 62
4.2.3 调度阶段的提交 62
4.2.4 任务的提交与执行 62
4.3 基于 IntelliJ IDEA 构建 Spark 应用程序 64
4.3.1 安装 IntelliJ IDEA 64
4.3.2 创建 Spark 应用程序 70
4.3.3 集群模式运行 Spark 应用程序 81

第 5 章 GraphX 87

5.1 GraphX 概述 87
5.2 GraphX 基本原理 89
5.2.1 图计算模型处理流程 89
5.2.2 GraphX 定义 90
5.2.3 GraphX 的特点 90
5.3 GraphX 设计与实现 91
5.3.1 弹性分布式属性图 91
5.3.2 图的数据模型 92
5.3.3 图的存储模型 94
5.3.4 GraphX 模型框架 97
5.4 GraphX 操作 97
5.4.1 创建图 97
5.4.2 基本属性操作 100
5.4.3 结构操作 102
5.4.4 转换操作 103
5.4.5 连接操作 105
5.4.6 聚合操作 106
5.5 GraphX 案例解析 107
5.5.1 PageRank 算法与案例解析 107
5.5.2 Triangle Count 算法与案例解析 110

第 6 章 Spark SQL 113

6.1 Spark SQL 概述 113
6.2 Spark SQL 逻辑架构 116
6.2.1 SQL 执行流程 116
6.2.2 Catalyst 117
6.3 Spark SQL CLI 117
6.3.1 硬软件环境 117
6.3.2 集群环境 118
6.3.3 结合 Hive 118
6.3.4 启动 Hive 118
6.4 DataFrame 编程模型 119
6.4.1 DataFrame 简介 119
6.4.2 创建 DataFrames 120
6.4.3 保存 DataFrames 126

6.5 DataFrame 常见操作 127
6.5.1 数据展示 127
6.5.2 常用列操作 128
6.5.3 过滤 131
6.5.4 排序 132
6.5.5 其他常见操作 134
6.6 基于 Hive 的学生信息管理系统的 SQL 查询案例与解析 137
6.6.1 Spark SQL 整合 Hive 137
6.6.2 构建数据仓库 138
6.6.3 加载数据 141
6.6.4 查询数据 142

第 7 章 Spark Streaming 146

7.1 Spark Streaming 概述 146
7.2 Spark Streaming 基础概念 147
7.2.1 批处理时间间隔 147
7.2.2 窗口时间间隔 148
7.2.3 滑动时间间隔 148
7.3 DStream 基本概念 149
7.4 DStream 的基本操作 150
7.4.1 无状态转换操作 150
7.4.2 有状态转换操作 152
7.4.3 输出操作 153
7.4.4 持久化操作 154
7.5 数据源 154
7.5.1 基础数据源 154
7.5.2 高级数据源 155
7.6 Spark Streaming 编程模式与案例分析 156
7.6.1 Spark Streaming 编程模式 156
7.6.2 文本文件数据处理案例（一） 157
7.6.3 文本文件数据处理案例（二） 160
7.6.4 网络数据处理案例（一） 164
7.6.5 网络数据处理案例（二） 171
7.6.6 stateful 应用案例 175
7.6.7 window 应用案例 180
7.7 性能考量 185
7.7.1 运行时间优化 185

7.7.2 内存使用与垃圾回收 186

第 8 章 Spark MLlib 187

8.1 Spark MLlib 概述 187
8.1.1 机器学习介绍 187
8.1.2 Spark MLlib 简介 189
8.2 MLlib 向量与矩阵 190
8.2.1 MLlib 向量 190
8.2.2 MLlib 矩阵 192
8.3 Spark MLlib 分类算法 196
8.3.1 贝叶斯分类算法 197
8.3.2 支持向量机算法 201
8.3.3 决策树算法 204
8.4 MLlib 线性回归算法 208
8.5 MLlib 聚类算法 212
8.6 MLlib 协同过滤 215

第1章

Spark 导论

半个多世纪以来，人类社会正由工业社会全面进入信息社会，其主要动力来自于以计算机技术、通信技术和控制技术为核心的现代化信息技术的飞速发展和扩展应用。信息继物质、能源之后，成为人类社会的第三大资源，可以说信息维系并推动着社会与经济的生存与发展。

近年来，互联网+概念的提出为信息技术的发展注入了更强劲的动力。所谓“互联网+”是互联网思维的进一步实践与拓展，它代表一种先进的生产力，推动经济形态不断演变，从而为实体经济的改革、发展和创新提供了广阔的网络平台。“互联网+”利用信息通信技术把互联网和包括传统行业在内的各行各业结合起来，例如互联网+金融衍生出互联网金融；互联网+零售衍生出电子商务；互联网+制造业衍生出工业 4.0 等。但这些并不是简单的两者相加，而是利用信息通信技术以及互联网平台让互联网与传统行业进行深度融合，创造新的发展生态。由此可见，互联网+、工业 4.0 等这些概念都与信息技术有着最密切和最直接的关系。信息技术的发展已经提升到国家战略层面这一前所未有的高度。

一般来说，信息既是对各种事物的特征和变化的反映，又是事物之间相互作用和联系的表征。人们通过信息来认识事物。信息以数据作为载体，数值、文字、语言、图形、图像等都是不同形式的数据。所以信息技术所处理的最主要对象就是数据。目前我们正处在一个数据爆炸的时代，大量涌现的智能手机、平板电脑、可穿戴设备及物联网设备每时每刻都在产生新的数据。据统计，有 90% 的数据是在过去短短两年内产生的，到 2020 年，将有 500 多亿台的互联设备产生 Zeta 字节级的数

据。在这一背景和趋势下，产业界将数据驱动的发展策略逐渐提升到前所未有的高度；在传媒、金融、电信等众多传统领域中，以往沉积的“垃圾”数据被重新审视，如何利用大数据分析技术“变废为宝”已经成为企业进行创新与盈利的一种重要模式；在学术界，国内外越来越多的高校和研究机构在云计算和大数据相关领域开展了深入的研究工作。然而在大数据时代，带来革命性改变的并非仅仅数据量本身，还有数据的种类、变化速度和价值密度。随着大数据特征的日益凸显，传统的信息技术已经难以准确高效地处理大数据背景下的数据采集、表示、存储、传输、处理等一系列问题。因此大数据相关技术是当前信息技术发展中亟需解决的关键技术。

综上所述，信息技术是推动未来社会发展的一种主流技术，信息技术处理的对象是数据，而当前的数据具有大数据特征，所以大数据技术成为了信息技术的重中之重。本书介绍的 Spark 以其卓越的性能，是目前大数据领域中最具潜力和影响力的处理引擎。

1.1 Spark 的发展

第一代大数据生态系统 Hadoop 已经非常成功，其采用 HDFS 实现分布式存储，使用 MapReduce 进行分布式计算。MapReduce 是一个简单通用和自动容错的批处理计算模型，不仅极大地简化了并行程序的开发过程，而且提高了程序执行的效率。然而在 MapReduce 任务内部，为了防止 Reduce 任务的失败，Map 通常会把结果存储在磁盘上。由于 MapReduce 每次都需要将中间数据写回磁盘，导致网络通信、磁盘 I/O 等消耗了大量的系统资源，尤其是在处理具有较多迭代次数的计算任务时，这一缺点尤为突出。因此 MapReduce 的运算性能仍难以满足面向大数据的交互式查询、迭代计算、流式计算等方面的需求。为了弥补 MapReduce 这一不足，涌现出了一系列专用的数据处理模型，例如 Storm、Impala、GraphLab 等。随着新模型的不断出现，不同类型的作业需要一系列不同的处理框架才能很好地完成，然而这无形中又增加了系统在部署、测试、运维等方面的成本。

针对 MapReduce 及各种专用数据处理模型在计算性能、集成性、部署运维等方面的问题，2009 年美国加洲大学伯克利分校开始研发全新的大数据处理框架，即 Spark。2010 年，Spark 实现开源。自从 2013 年 Spark 进入 Apache 的孵化器项目后，发生了翻天覆地的变化。2014 年初，Spark 成为了 Apache 排名第三的顶级项目，其发展势头更加迅猛，一个多月左右就会发布一个小版本，两三个月左右会发布一个大版本，2015 年 6 月份发布了 1.4.0，2015 年 9 月份发布了 1.5.0，至本书作者执笔时已经发布到 2.0。

Spark 的迅猛发展和其突出的计算性能引起了大数据处理相关领域的广泛关注，使得其迅速成为了业内一门具有强劲竞争力的热门技术。在短短的三年时间里，世界各地开设了多次 Spark 主题峰会，反映出 Spark 技术的前沿性与火爆的发展势头。在 2014 年的 Spark 峰会上，Hadoop 三大发行商均声称未来将会把精力投入到 Spark 的研究中；Yahoo 有全世界最强大的 Hadoop 集群（Hadoop 的 70%是由 Yahoo 贡献的），但早在几年前 Yahoo 便已经开启了 Spark 的研发工作；Intel、IBM 等大公司纷纷宣布其产品支持 Spark；亚马逊完全基于 Spark 搭建了云服务平台；Google、Facebook 也陆续开展了转向 Spark 框架的研发工作。

同样也是在2014年Spark在中国的发展达到了一个前所未有的火爆状态。国内许多重量级的数据企业纷纷搭建了自己的Spark集群。例如2015年百度搭建了一个大规模的Spark集群，其中最大单集群规模达上千台节点，包含了数万个核心和上百 TB 的内存，与此同时公司内部还运行着大量的小型Spark集群。淘宝的推荐系统已经用Spark取代了部分的MapReduce。腾讯通过Spark对数据实时采集、算法实时训练、系统实时预测，进而实现了大数据背景下的精准推荐。目前国内最大规模的Spark集群来自于腾讯，其中包含了8000个节点，而最大的单个Job则来自于阿里巴巴。虽然上述集群中节点数目看似不多，但是1000个节点的Spark集群，其性能相当于包含5000个节点的Hadoop集群。

1.2 什么是Spark

Apache官网对Spark定义如下：

“Apache Spark is a fast and general engine for large-scale data processing”

由此可见，Spark 是一个快速通用的大规模数据处理引擎。Spark的功能看似与Hadoop相同，但是两者却有着明显的区别。Hadoop是一个开源分布式计算平台，是第一代大数据生态系统，也是目前应用最为广泛的大数据生态系统。Hadoop 以分布式文件系统 HDFS 和分布式计算 MapReduce 为核心，为用户提供了系统底层细节透明的分布式基础框架。HDFS 的高容错性、高伸缩性等优点允许用户将Hadoop部署在低廉的硬件上；MapReduce分布式编程模型允许用户在不了解分布式系统底层细节的情况下开发并行应用程序。因此用户利用Hadoop能够轻松地组织计算机资源，从而搭建自己的分布式计算平台，并且可以充分利用集群的计算和存储能力完成海量数据的处理。Spark作为大数据处理引擎与Hadoop有着密切的联系，但是两者并非是一个层面的概念。Hadoop不仅有数据的处理还有数据的存储，而Spark仅仅是大数据处理框架。因此Spark其实与Hadoop上的MapReduce是一个层面的概念。目前的Hadoop已经趋于完善，其中的Yarn与HDFS非常经典，已经成为了业内大数据存储和分布式资源管理的标准，在未来短时间内难以被轻易取代。很多资料均预言 Spark将取代Hadoop，其实是指取代Hadoop中的MapReduce。

1.3 Spark主要特征

Spark是一种基于内存的、分布式的、大数据处理框架，它与Hadoop上的MapReduce是一个层面上的概念，这意味着两者在诸多方面存在着竞争与可比性。本节将通过与MapReduce的对比分析来介绍Spark的主要特征。

1.3.1 快速

面向磁盘的MapReduce受限于磁盘读/写性能和网络I/O性能的约束，在处理迭代计算、实时计

算、交互式数据查询等方面并不高效，但是这些却在图计算、数据挖掘和机器学习等相关应用领域中非常常见。针对这一不足，将数据存储在内存中并基于内存进行计算是一个有效的解决途径。Spark 是面向内存的大数据处理引擎，这使得 Spark 能够为多个不同数据源的数据提供近乎实时的处理性能，适用于需要多次操作特定数据集的应用场景。

在相同的实验环境下处理相同的数据，若在内存中运行，那么 Spark 要比 MapReduce 快 100 倍，如图 1.1 所示；在磁盘中运行时 Spark 要比 MapReduce 快 10 倍，如图 1.2 所示。综合各种实验表明，处理迭代计算问题 Spark 要比 MapReduce 快 20 多倍，计算数据分析类报表的速度可提高 40 多倍，能够在 5~7 秒的延时内交互式扫描 1TB 数据集。

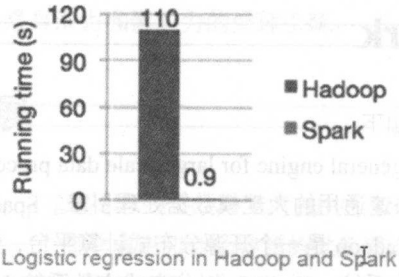


图 1.1　基于内存 Spark 与 MapReduce 执行逻辑回归的性能对比

排序问题是最考验系统性能的问题之一。图 1.2 是 Spark 与 MapReduce 对相同的 100TB 数据样本排序的性能对比。在实验中，MapReduce 用了 2100 台节点，用时 72 分钟；而 Spark 仅用 207 台节点，是前者的 1/10，用时 23 分钟，是前者的 1/3。

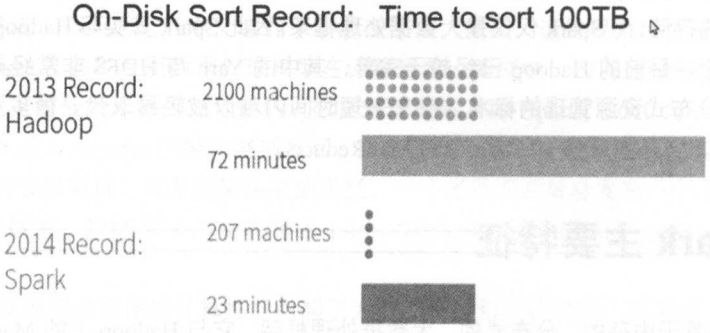


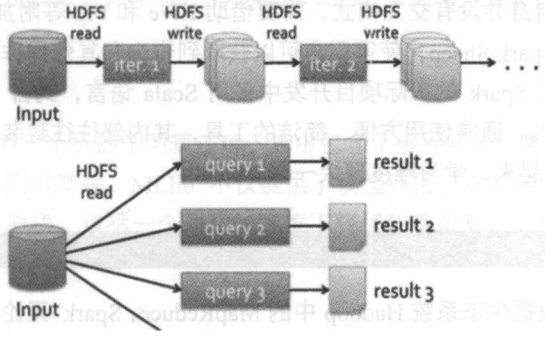
图 1.2　基于磁盘 Spark 与 MapReduce 对 100T 数据排序的性能对比

Spark 与 MapReduce 相比在计算性能上有如此显著的提升，主要得益于以下两方面。

1. Spark 是基于内存的大数据处理框架

Spark 既可以在内存中处理一切数据，也可以使用磁盘来处理未全部装入到内存中的数据。由于

内存与磁盘在读/写性能上存在巨大的差距，因此 CPU 基于内存对数据进行处理的速度要快于磁盘数倍。然而 MapReduce 对数据的处理是基于磁盘展开的。一方面，MapReduce 对数据进行 Map 操作后的结果要写入磁盘中，而且 Reduce 操作也是在磁盘中读取数据，另一方面，分布式环境下不同物理节点间的数据通过网络进行传输，网络性能使得该缺点进一步被放大。因此磁盘的读/写性能、网络传输性能成为了基于MapReduce大数据处理框架的瓶颈。图 1.3 为MapReduce数据处理流程示意图。



（a） MapReduce 处理流程

图 1.3 MapReduce 数据处理流程示意图

2. Spark 具有优秀的作业调度策略

Spark 中使用了有向无环图（Directed Acyclic Graph，DAG）这一概念。一个 Spark 应用由若干个作业构成，首先 Spark 将每个作业抽象成一个图，图中的节点是数据集，图中的边是数据集之间的转换关系；然后 Spark 基于相应的策略将 DAG 划分出若干个子图，每个子图称为一个阶段，而每个阶段对应一组任务；最后每个任务交由集群中的执行器进行计算。借助于 DAG，Spark 可以对应用程序的执行进行优化，能够很好地实现循环数据流和内存计算。

1.3.2 简洁易用

Spark 不仅计算性能突出，在易用性方面也是其他同类产品难以比拟的。一方面，Spark 提供了支持多种语言的 API，如 Scala、Java、Python、R 等，使得用户开发 Spark 程序十分方便。另一方面，Spark 是基于 Scala 语言开发的，由于 Scala 是一种面向对象的、函数式的静态编程语言，其强大的类型推断、模式匹配、隐式转换等一系列功能结合丰富的描述能力使得 Spark 应用程序代码非常简洁。Spark 的易用性还体现在其针对数据处理提供了丰富的操作。在使用 MapReduce 开发应用程序时，通常用户关注的重点与难点是如何将一个需求 Job（作业）拆分成 Map 和 Reduce。由于 MapReduce 中仅为数据处理提供了两个操作，即 Map 和 Reduce，因此系统开发人员需要解决的一个难题是如何把数据处理的业务逻辑合理有效地封装在对应的两个类中。与之相对比，Spark 提供了 80 多个针对数据处理的基本操作，如 map、flatMap、reduceByKey、filter、cache、collect、textFile 等，这使得用户基于 Spark 进行应用程序开发非常简洁高效。以分词统计为例，虽然 MapReduce 固

定的编程模式极大地简化了并行程序开发，但是代码至少几十行；若换成 Spark，其核心代码最短仅需一行，如示例 1-1 所示，极大地提高了应用程序开发效率。

示例 1-1　基于 Spark 的 WordCount 程序核心代码

```
    sc.textFile("hdfs://master:8020/user/dong/Spark/wc.input").flatMap(_.spl
it("")).map((_,1)).reduceByKey(_ + _).collect
```

此外，MapReduce 自身并没有交互模式，需要借助 Hive 和 Pig 等附加模块。Spark 则提供了一种命令行交互模式，即 Spark Sheep，使得用户可以获取到查询和其他操作的即时反馈。

但需要注意的是，在 Spark 的实际项目开发中多用 Scala 语言，约占 70%；其次是 Java，约占 20%；而 Python 约占 10%。通常使用方便、简洁的工具，其内部往往封装了更为复杂的机理，因此 Scala 与 Java 等语言比较起来，学习难度要大一些。

1.3.3　通用

相对于第一代的大数据生态系统 Hadoop 中的 MapReduce，Spark 无论是在性能还是在方案的统一性方面，都有着极大的优势。Spark 框架包含了多个紧密集成的组件，如图 1.4 所示。位于底层的是 Spark Core，其实现了 Spark 的作业调度、内存管理、容错、与存储系统交互等基本功能，并针对弹性分布式数据集提供了丰富的操作。在 Spark Core 的基础上，Spark 提供了一系列面向不同应用需求的组件，主要有 Spark SQL、Spark Streaming、MLlib、GraphX。

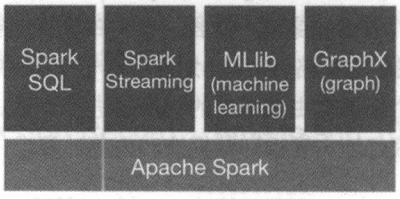


图 1.4　Spark 软件栈

1. Spark SQL

Spark SQL 是 Spark 用来操作结构化数据的组件。通过 Spark SQL，用户可以使用 SQL 或者 Apache Hive 版本的 SQL 方言（HQL）来查询数据。Spark SQL 支持多种数据源类型，例如 Hive 表、Parquet 以及 JSON 等。Spark SQL 不仅为 Spark 提供了一个 SQL 接口，还支持开发者将 SQL 语句融入到 Spark 应用程序开发过程中，无论是使用 Python、Java 还是 Scala，用户可以在单个的应用中同时进行 SQL 查询和复杂的数据分析。由于能够与 Spark 所提供的丰富的计算环境紧密结合，Spark SQL 得以从其他开源数据仓库工具中脱颖而出。Spark SQL 在 Spark 1.0 中被首次引入。在 Spark SQL 之前，美国加州大学伯克利分校曾经尝试修改 Apache Hive 以使其运行在 Spark 上，进而提出了组件 Shark。然而随着 Spark SQL 的提出与发展，其与 Spark 引擎和 API 结合得更加紧密，使得 Shark 已经被 Spark SQL 所取代。

2. Spark Streaming

众多应用领域对实时数据的流式计算有着强烈的需求，例如网络环境中的网页服务器日志或是由用户提交的状态更新组成的消息队列等，这些都是实时数据流。Spark Streaming 是 Spark 平台上针对实时数据进行流式计算的组件，提供了丰富的处理数据流的 API。由于这些 API 与 Spark Core 中的基本操作相对应，因此开发者在熟知 Spark 核心概念与编程方法之后，编写 Spark Streaming 应用程序会更加得心应手。从底层设计来看，Spark Streaming 支持与 Spark Core 同级别的容错性、吞吐量以及可伸缩性。

3. MLlib

MLlib 是 Spark 提供的一个机器学习算法库，其中包含了多种经典、常见的机器学习算法，主要有分类、回归、聚类、协同过滤等。MLlib 不仅提供了模型评估、数据导入等额外的功能，还提供了一些更底层的机器学习原语，包括一个通用的梯度下降优化基础算法。所有这些方法都被设计为可以在集群上轻松伸缩的架构。

4. GraphX

GraphX 是 Spark 面向图计算提供的框架与算法库。GraphX 中提出了弹性分布式属性图的概念，并在此基础上实现了图视图与表视图的有机结合与统一；同时针对图数据处理提供了丰富的操作，例如取子图操作 subgraph、顶点属性操作 mapVertices、边属性操作 mapEdges 等。GraphX 还实现了与 Pregel 的结合，可以直接使用一些常用图算法，如 PageRank、三角形计数等。

上述这些 Spark 核心组件都以 jar 包的形式提供给用户，这意味着在使用这些组件时，与 MapReduce 上的 Hive、Mahout、Pig 等组件不同，无需进行复杂烦琐的学习、部署、维护和测试等一系列工作，用户只要搭建好 Spark 平台便可以直接使用这些组件，从而节省了大量的系统开发与运维成本。将这些组件放在一起，就构成了一个 Spark 软件栈。基于这个软件栈，Spark 提出并实现了大数据处理的一种理念——“一栈式解决方案（one stack to rule them all）”，即 Spark 可同时对大数据进行批处理、流式处理和交互式查询，如图 1.5 所示。借助于这一软件栈用户可以简单而低耗地把各种处理流程综合在一起，充分体现了 Spark 的通用性。

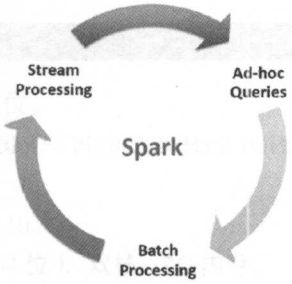


图 1.5 Spark 面向大数据的综合处理示意图

1.3.4 多种运行模式

Spark 支持多种运行模式：本地 local 运行模式、分布式运行模式。Spark 集群的底层资源可以借助于外部的框架进行管理，目前 Spark 对 Mesos 和 Yarn 提供了相对稳定的支持。在实际生产环境中，中小规模的 Spark 集群通常可满足一般企业绝大多数的业务需求，而在搭建此类集群时推荐采用 Standalone 模式（不采用外部的资源管理框架）。该模式使得 Spark 集群更加轻量级。

Spark on Yarn 模式：在这一模式下，Spark 作为一个提交程序的客户端将 Spark 任务提交到 Yarn 上，然后通过 Yarn 来调度和管理 Spark 任务执行过程中所需的资源。在搭建此模式的 Spark 集群过程中，需要先搭建 Yarn 集群，然后将 Spark 作为 Hadoop 中的一个组件纳入到 Yarn 的调度管理下，这样将更有利于系统资源的共享。

Spark on Mesoes 模式：Spark 和资源管理框架 Mesos 相结合的运行模式。Apache Mesos 与 Yarn 类似，能够将 CPU、内存、存储等资源从计算机的物理硬件中抽象地隔离出来，搭建了一个高容错、弹性配置的分布式系统。Mesos 同样也采用 Master/Slave 架构，并支持粗粒度模式和细粒度模式两种调度模式。

Spark Standalone 模式：该模式是不借助于第三方资源管理框架的完全分布式模式。Spark 使用自己的 Master 进程对应用程序运行过程中所需的资源进行调度和管理。对于中小规模的 Spark 集群首选 Standalone 模式。本书将着重介绍 Standalone 模式下 Spark 集群的搭建方法。

第2章

Spark 集群部署

搭建 Spark 平台是开发 Spark 应用程序的第一步，也是至关重要的一步。针对不同的应用场景，Spark 集群的运行模式有多种，本章将详细介绍 Standalone 模式下 Spark 集群的部署。对 Local、Mesos、Yarn 模式下的部署不作具体介绍，如有需要，请读者自行参考相关文档。

2.1 运行环境说明

2.1.1 软硬件环境

主机操作系统：Windows 8，64 位。

处理器：Intel(R) Core(TM) i5-5200U CPU@ 2.2GHz 2.2GHz。

内存：8.0 GB。

虚拟软件：VMware Workstation 10。

虚拟机操作系统：CentOS6.5（64 位），双核，2G 内存。

虚拟机运行环境：JDK1.7（64 位），Hadoop2.5.2，Scala2.10.4，Spark1.4.0。

2.1.2 集群网络环境

本书演示所用集群共包含三个节点，所有节点均是 CentOS6.5 64 位系统，防火墙均禁用，每个节点上创建了一个 dong 用户，用户的主目录是/home/dong，节点之间可以免密码 SSH 访问，节点 IP 地址和主机名分布如表 2.1 所示。

表 2.1 集群环境表

序　号	IP 地址	机器名	类　型	用户名
1	192.168.149.132	master	NameNode/Master	dong
2	192.168.149.133	slave1	DataNode/Worker	dong
3	192.168.149.134	slave2	DataNode/Worker	dong

其中 NameNode 表示 Hadoop 集群中的主节点，DataNode 表示 Hadoop 集群中的从节点，Master 表示 Spark 集群中的主节点，Worker 表示 Spark 集群中的从节点。

2.2 安装 VMware Workstation 11

对于 Spark 初学者，在软硬件资源有限的条件下搭建一个高性能、包含多物理节点的 Spark 集群比较困难。为方便讲解与演示，本书首先使用 VMware Workstation 构建的实验环境模拟分布式集群。VMware Workstation 虚拟机是一个在 Windows 或 Linux 上运行的应用程序，它可以模拟标准的 PC 环境。这个环境和真实的计算机一样，都有芯片组、CPU、内存、显卡、声卡、网卡、软驱、硬盘、光驱、串口、并口、USB 控制器、SCSI 控制器等设备，提供这个应用程序的窗口就是虚拟机的显示器。在使用上，这台虚拟机和真正的物理主机没有太大的区别，都需要分区、格式化、安装操作系统、安装应用程序和软件。总之，在虚拟机上的一切操作都跟一台真正的计算机一样。VMware Workstation11 安装步骤如下：

第 1 步，打开 VMware Workstation 11，界面如图 2.1 所示。

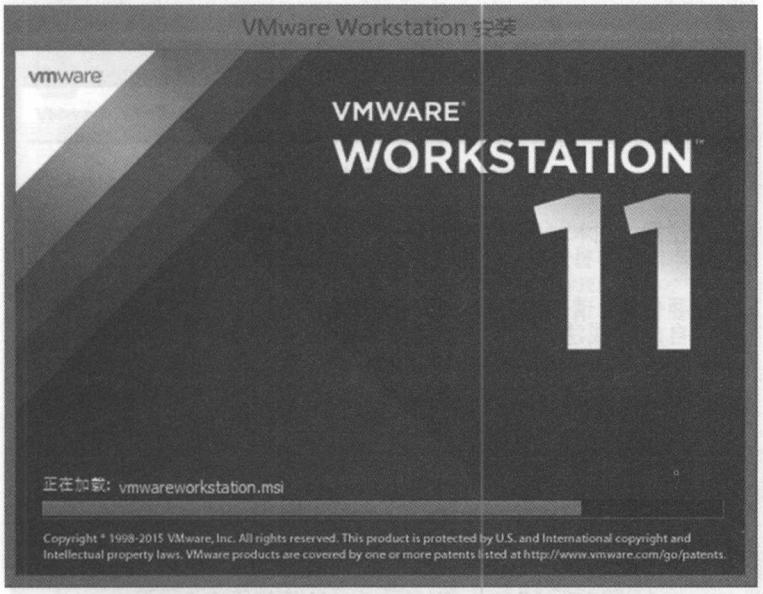


图 2.1　VMware Workstation 11 安装第 1 步界面

第 2 步，等待安装软件检测和解压以后，出现如图 2.2 所示界面，单击“下一步”按钮即可。

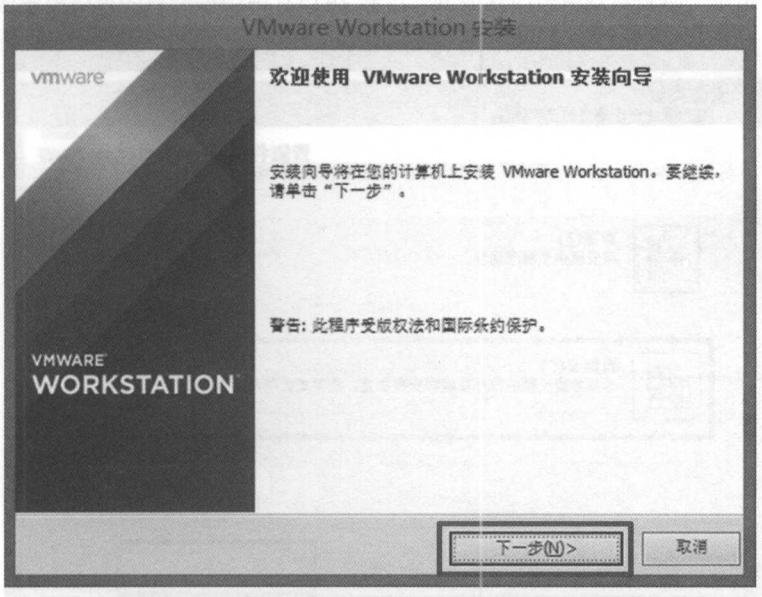


图 2.2　VMware Workstation 11 安装第 2 步界面

第 3 步，选择同意选项，并单击“下一步”按钮，如图 2.3 所示。

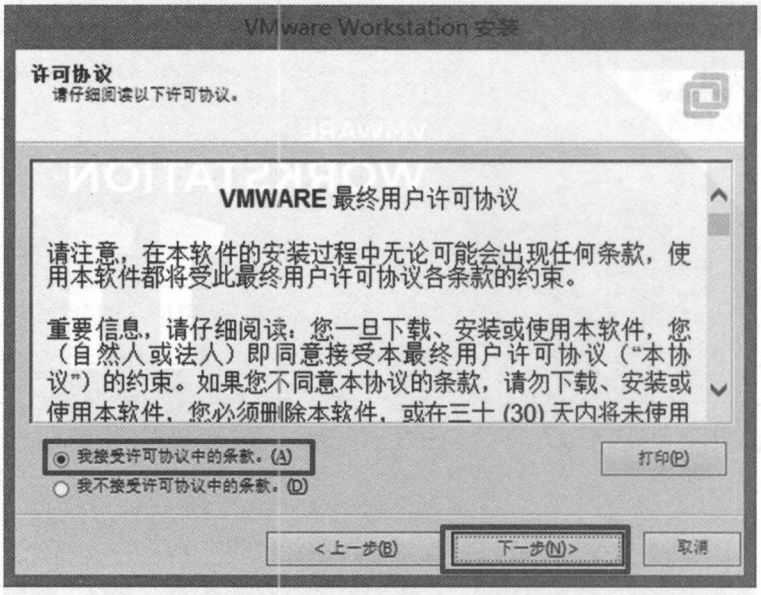


图 2.3　VMware Workstation 11 安装第 3 步界面

第 4 步，用户根据自己实际情况选择相应的安装类型。此处我们选择“自定义”安装，然后单击“下一步”按钮，如图 2.4 所示。

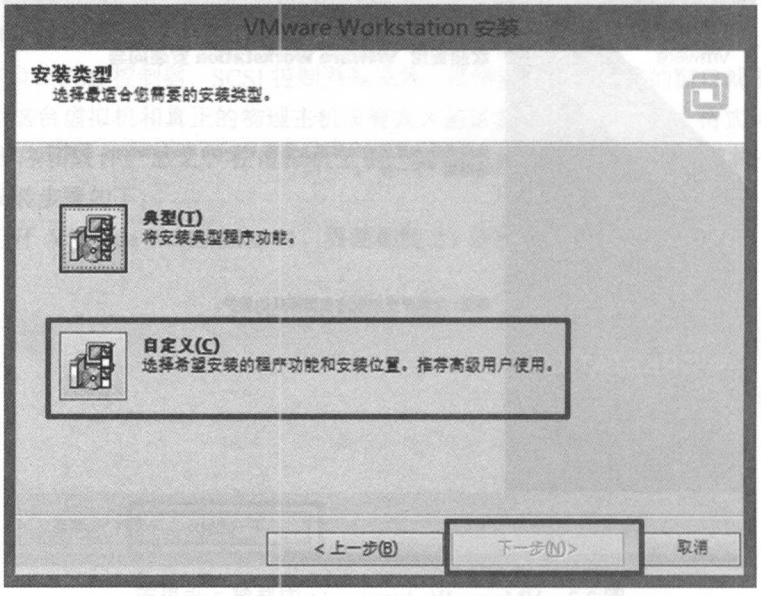


图 2.4　VMware Workstation 11 安装第 4 步界面

第 5 步，用户根据自己的实际情况选择相应的功能模块。此处我们选择所有功能模块，并修改

了安装路径，如图 2.5 所示。

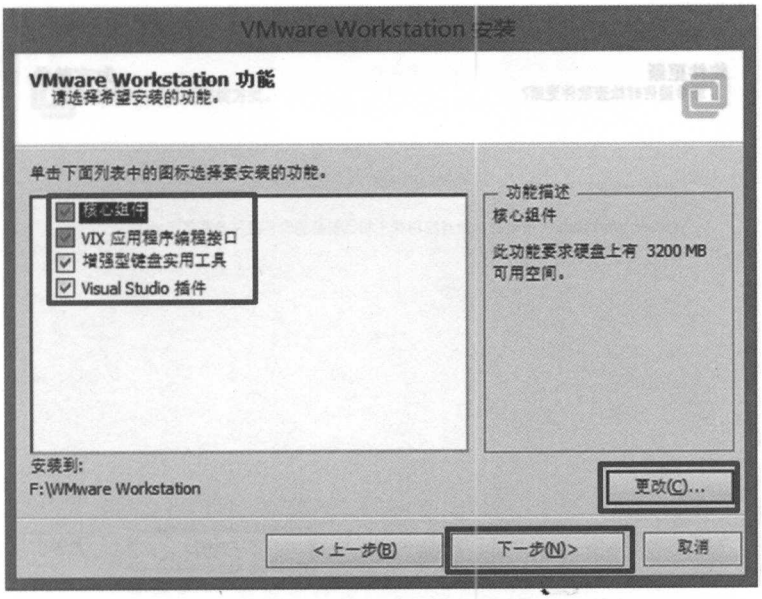


图 2.5　VMware Workstation 11 安装第 5 步界面

第 6 步，配置虚拟机的存储路径和 HTTPS 端口，此处我们选用默认选项，单击“下一步”按钮如图 2.6 所示。

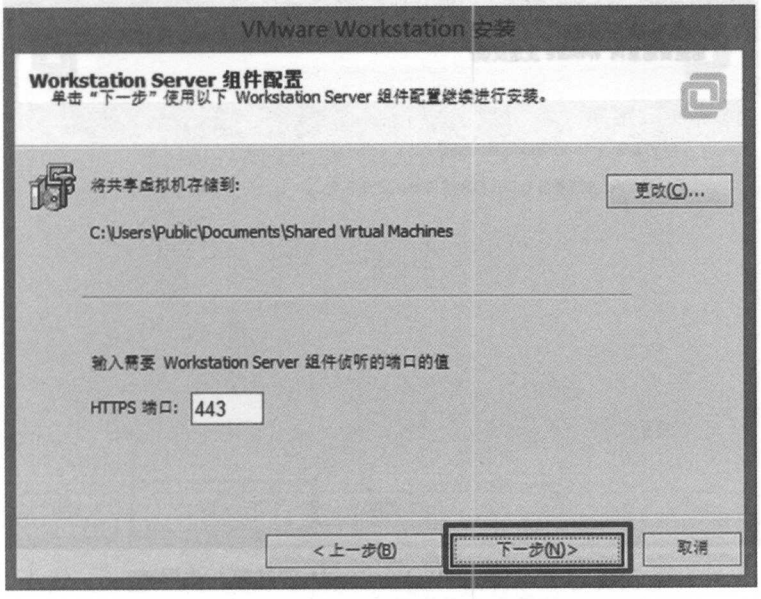


图 2.6　VMware Workstation 11 安装第 6 步界面

第 7 步，用户可根据实际情况选择对“软件更新”进行配置。

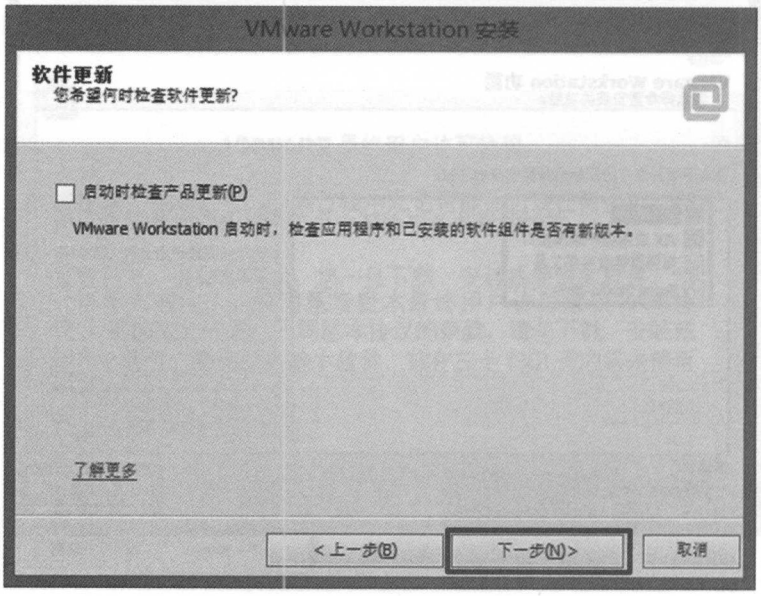


图 2.7　VMware Workstation 11 安装第 7 步界面

第 8 步，用户可根据实际情况对“用户体验改进计划”进行配置。

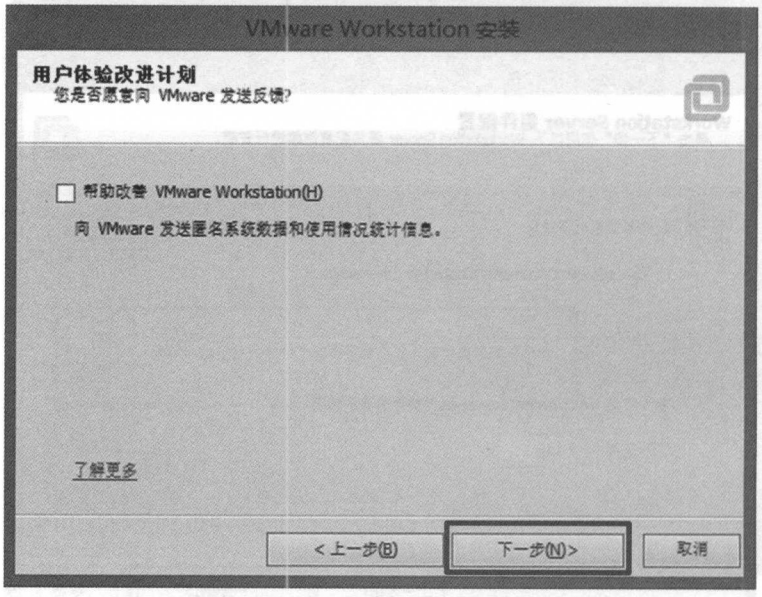


图 2.8　VMware Workstation 11 安装第 8 步界面

第 9 步，快捷方式配置，此处我们选择默认配置，单击“下一步”按钮，如图 2.9 所示。

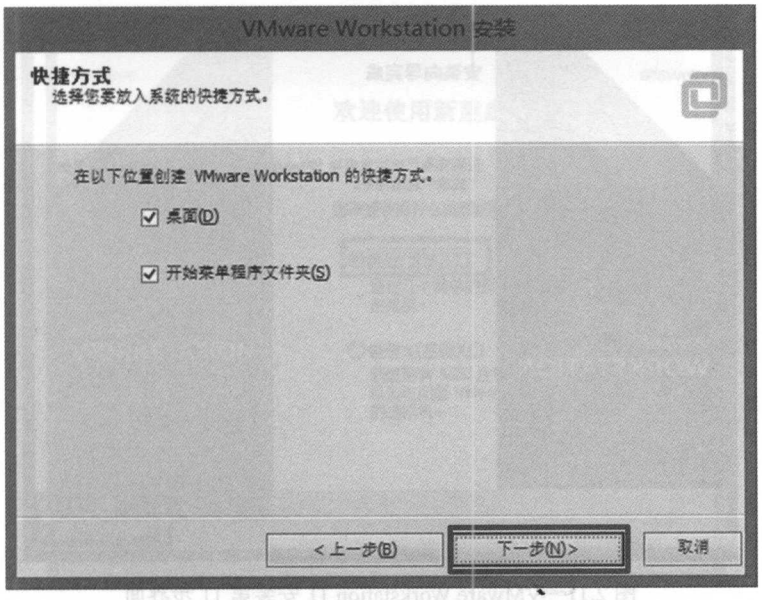


图 2.9　VMware Workstation 11 安装第 9 步界面

第 10 步，进行安装。单击“继续”按钮，如图 2.10 所示。

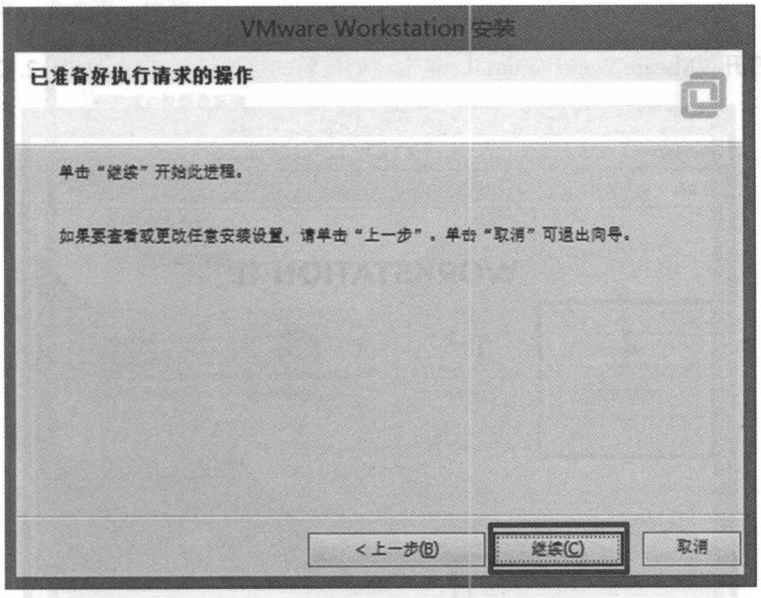


图 2.10　VMware Workstation 11 安装第 10 步界面

第 11 步，输入注册码，完成相应安装，如图 2.11 所示。

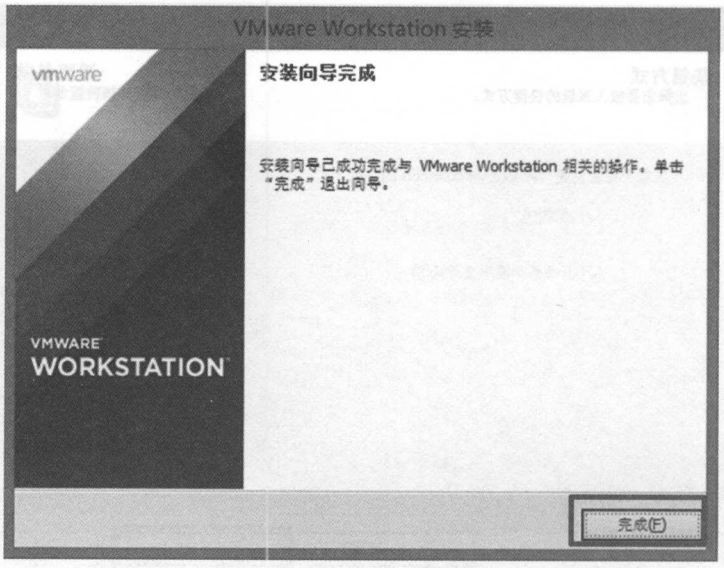


图 2.11　VMware Workstation 11 安装第 11 步界面

2.3 安装 CentOS 6

第 1 步，打开 VMware Workstation11，单击“创建新的虚拟机”按钮，如图 2.12 所示。

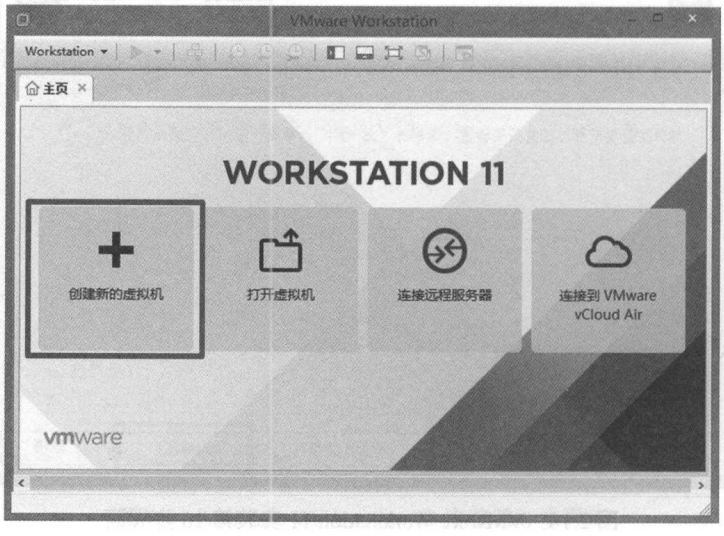


图 2.12　CentOS 安装第 1 步界面

第 2 步，选择“典型”选项，单击“下一步”按钮，如图 2.13 所示。

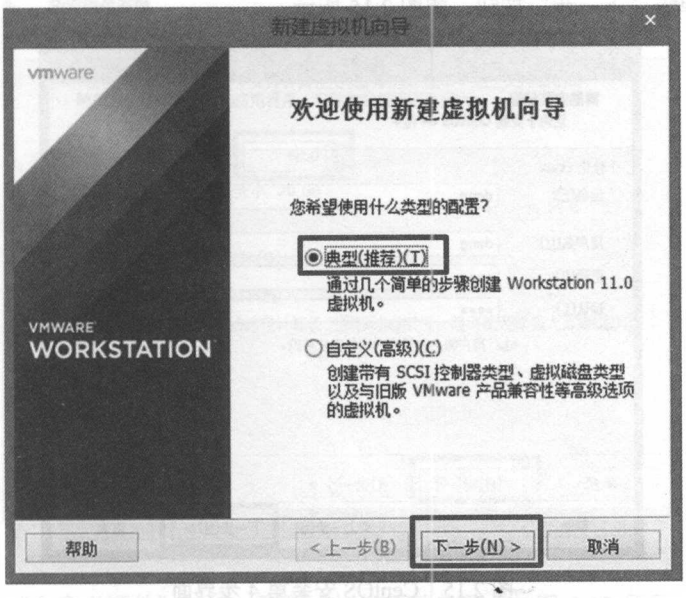


图 2.13　CentOS 安装第 2 步界面

第 3 步，选择“安装程序光盘映像文件”选项，选择指定的 CentOS 系统的.iso 文件，如图 2.14 所示，然后单击“下一步”按钮。

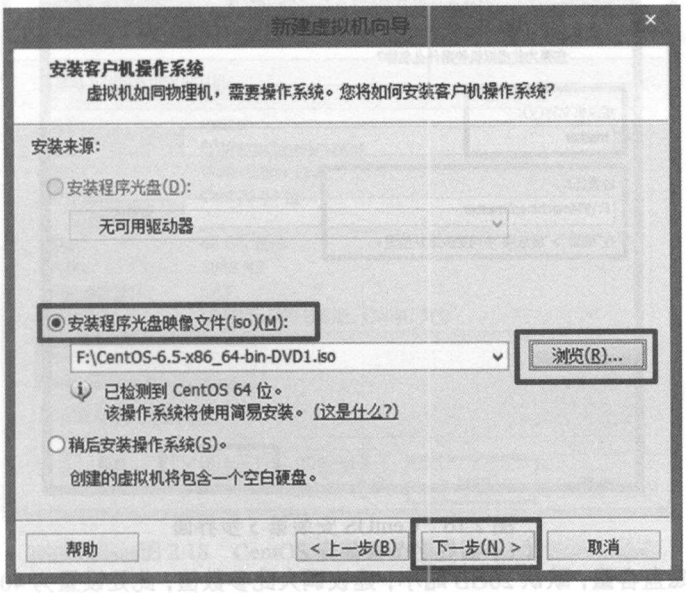


图 2.14　CentOS 安装第 3 步界面

第 4 步，填写简易安装信息，为方便演示，此处将“全名”、“用户名”、“密码”、“确认”均设置为 dong，然后单击“下一步”按钮，如图 2.15 所示。

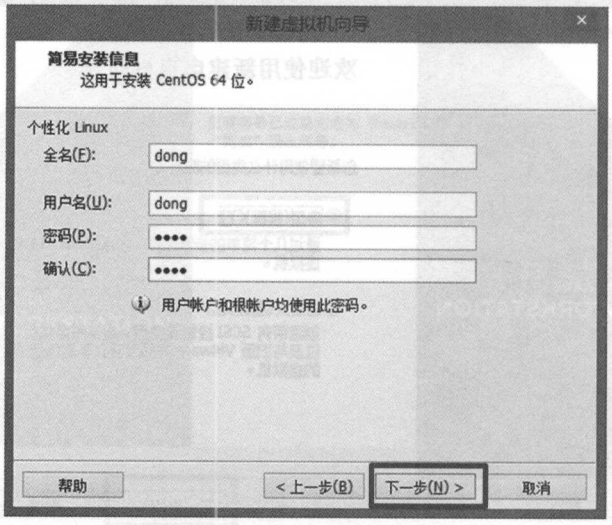


图 2.15　CentOS 安装第 4 步界面

第 5 步，设置“虚拟机名称”，此处设为 master，并为虚拟机选择安装位置，然后单击“下一步”按钮，如图 2.16 所示。

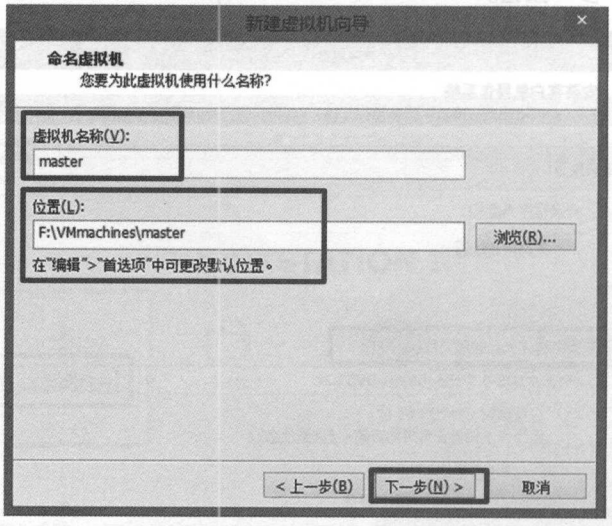


图 2.16　CentOS 安装第 5 步界面

第 6 步，指定磁盘容量，默认 20GB 略小，建议调大此参数值，此处设置为 40GB，然后单击“下一步”按钮，如图 2.17 所示。

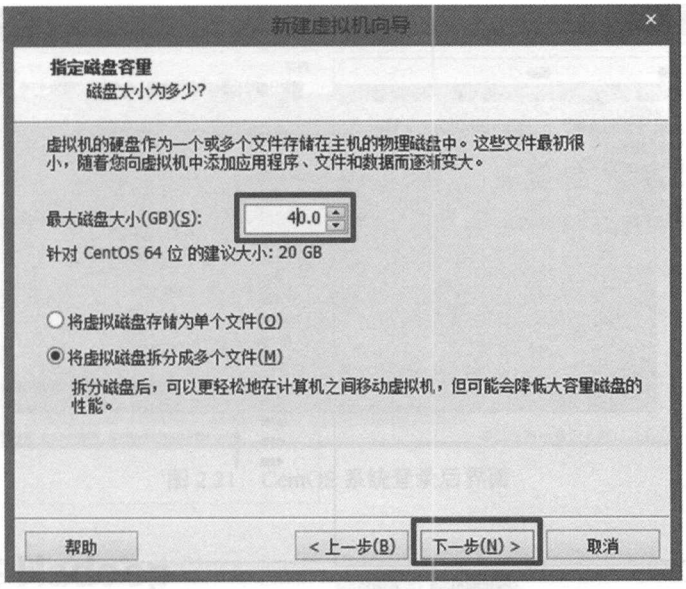


图 2.17　CentOS 安装第 6 步界面

第 7 步，单击“自定义硬件”按钮，配置虚拟机硬件参数，如图 2.18 所示。然后单击“完成”按钮。具体的硬件参数设置，如图 2-19 所示。

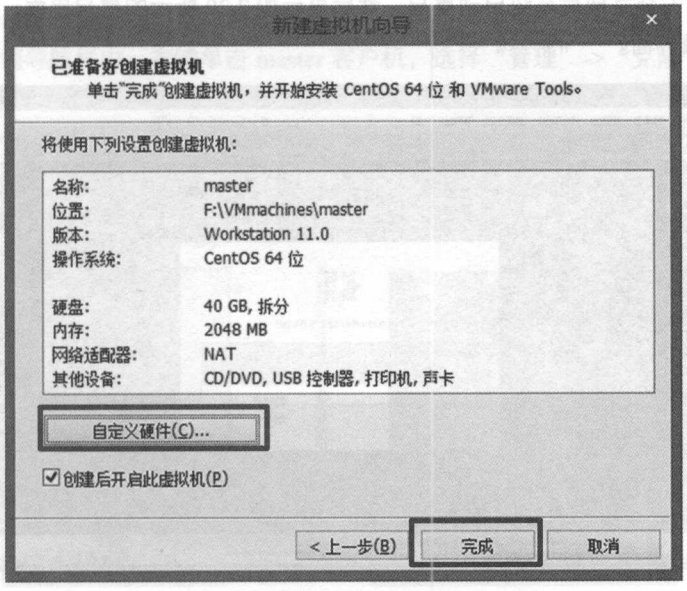


图 2.18　CentOS 安装第 7 步界面（1）

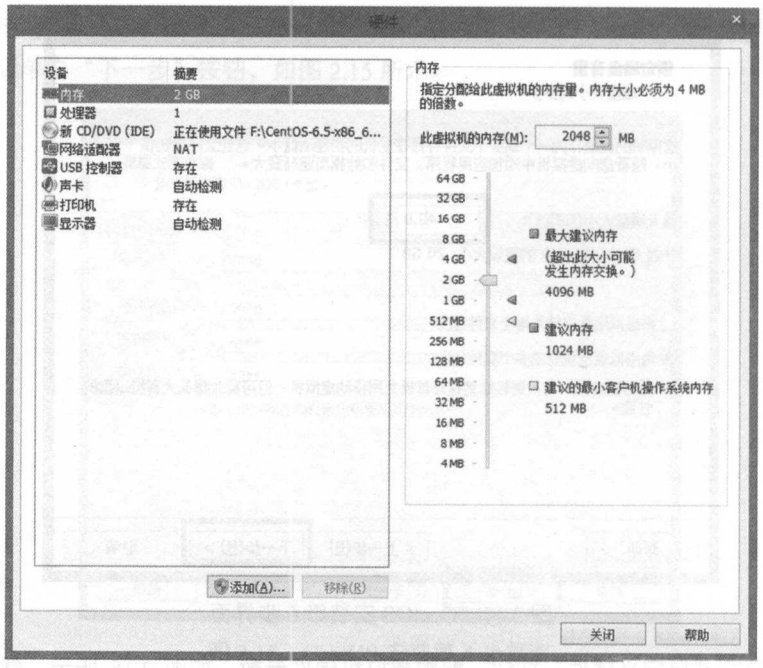


图 2.19　CentOS 安装第 7 步界面（2）

第 8 步，等待安装完成后系统自动重启，然后显示如 2.20 所示的重启界面；输入密码，显示如图 2.21 所示的登录界面。

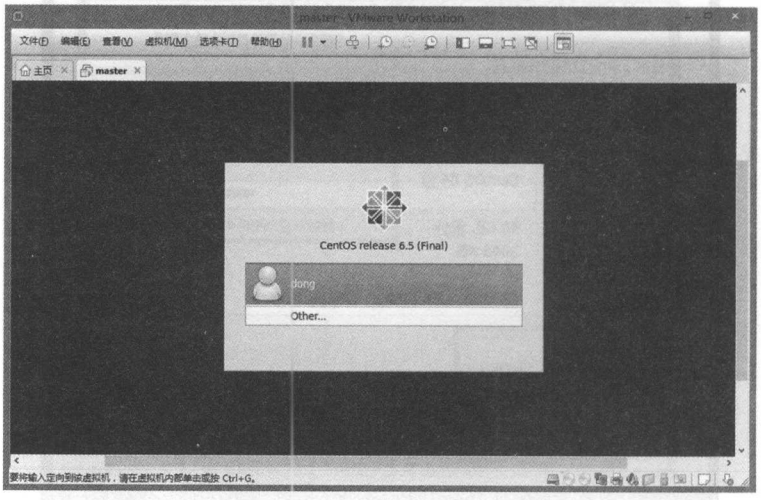


图 2.20　CentOS 安装完成后重启界面

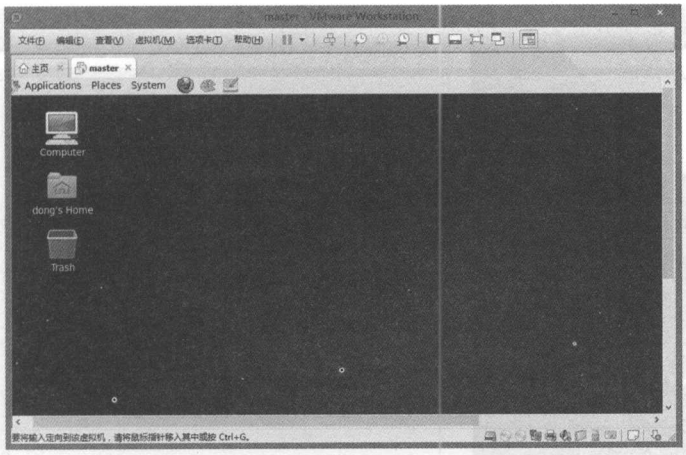


图 2.21　CentOS 系统登录后界面

2.4 安装 Hadoop

2.4.1 克隆并启动虚拟机

第 1 步，若 master 已经开启，首先将其关闭。

第 2 步，在左侧导航栏中，右键单击 master 客户机，选择“管理”->“克隆”选项，如图 2.22 和图 2-23 所示。打开“克隆虚拟机向导”、单击“下一步”按钮。

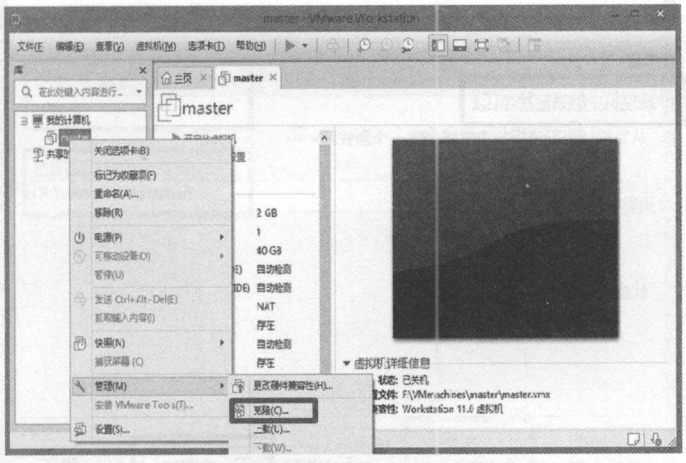


图 2.22　克隆 master 步骤 2 界面（1）

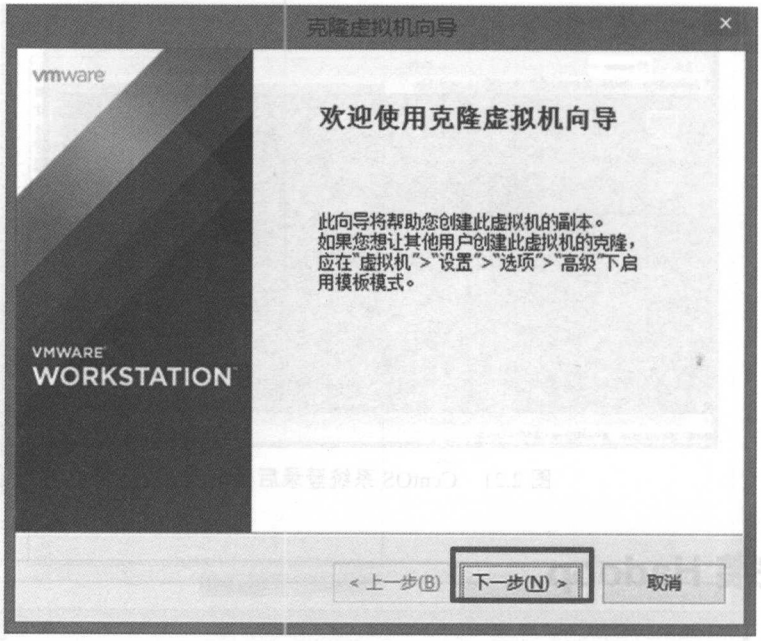


图 2.23　克隆 master 步骤 2 界面（2）

第 3 步，配置克隆源，然后单击“下一步”按钮，如图 2.24 所示。

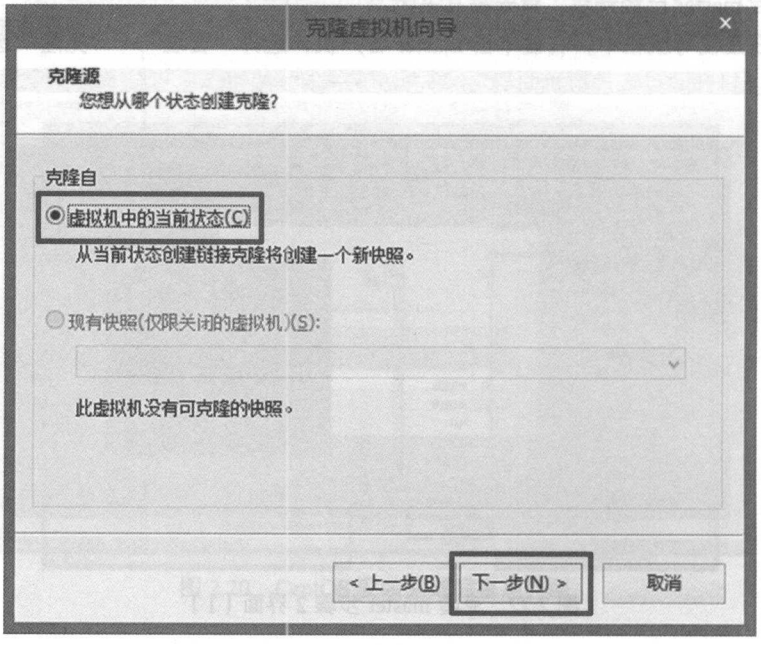


图 2.24　克隆 master 步骤 3 界面

第 4 步，配置克隆类型，选择“创建完整克隆”，然后单击“下一步”按钮，如图 2.25 所示。

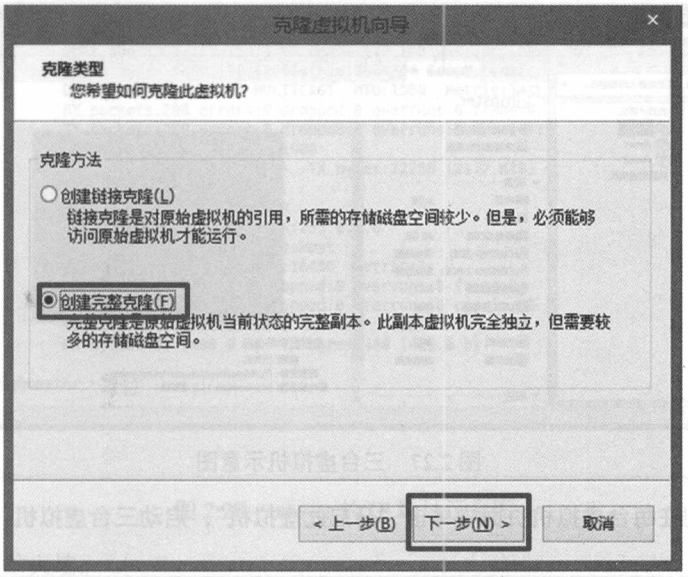


图 2.25　克隆 master 步骤 4 界面

第 5 步，配置克隆出的新虚拟机名称与位置，此处名称设为 slave1，并存放至 F 盘相应目录下，如图 2.26 所示。

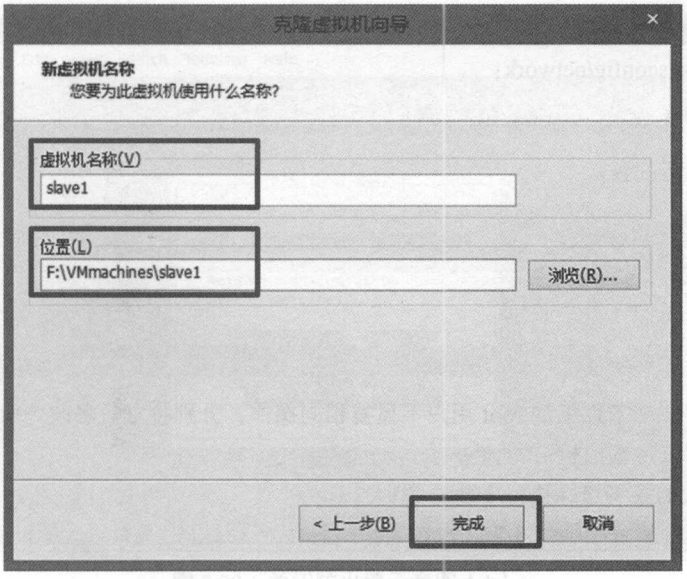


图 2.26　克隆 master 步骤 5 界面

第 6 步，与克隆 slave1 步骤类似，再克隆出新的虚拟机 slave2，如图 2.27 所示。

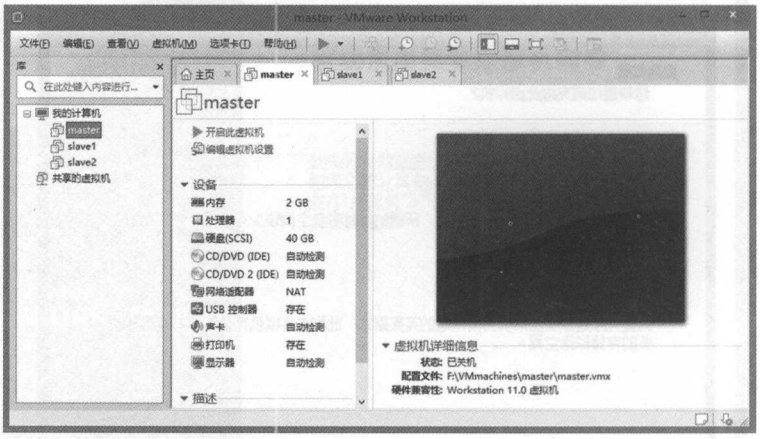


图 2.27　三台虚拟机示意图

第 7 步，分别在每台虚拟机页面中单击“开启此虚拟机”，启动三台虚拟机。

2.4.2　网络基本配置

第 1 步，配置三台虚拟机主机名称。

首先在 master 节点上切换至 root 用户，密码为 dong。

```
$ su - root
```

然后打开/etc/sysconfig/network：

```
$ gedit /etc/sysconfig/network
```

对文件进行如下配置：

```
NETWORKING=yes
HOSTNAME=master
```

使配置生效：

```
$ hostname master
```

在 slave1 与 slave2 节点上的 root 用户下重复相同操作，分别将主机名改为 slave1、slave2。

第 2 步，查看三台虚拟机 IP 地址。

在 master 节点的控制终端执行下面命令：

```
$ ifconfig
```

执行后如图 2.28 所示。

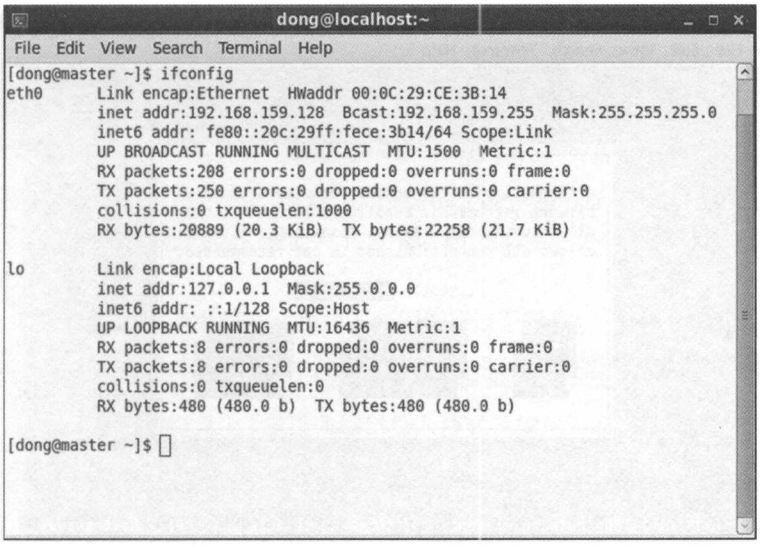


图 2.28　master 的 IP 地址示意图

第 3 步，关闭防火墙。

分别在 master、slave1、slave2 节点中进行以下操作。

```
$ setup
```

然后选择如图 2.29 ~ 图 2.31 中所示的操作。

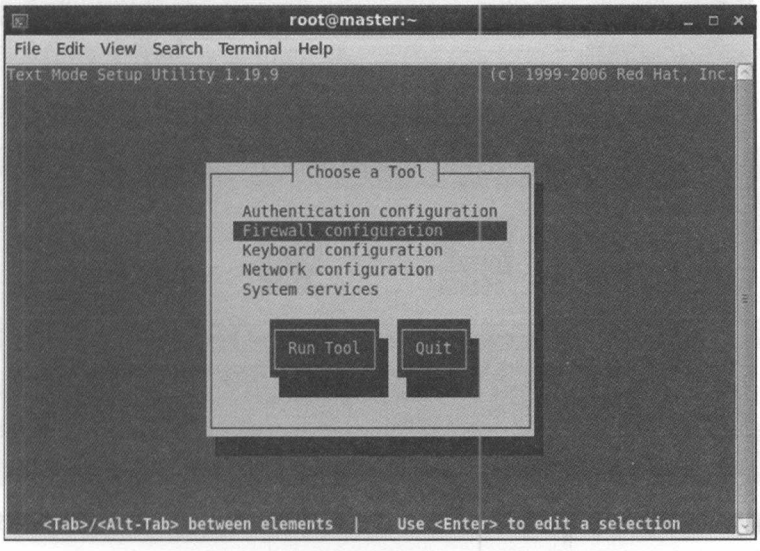


图 2.29　关闭防火墙示意图（1）

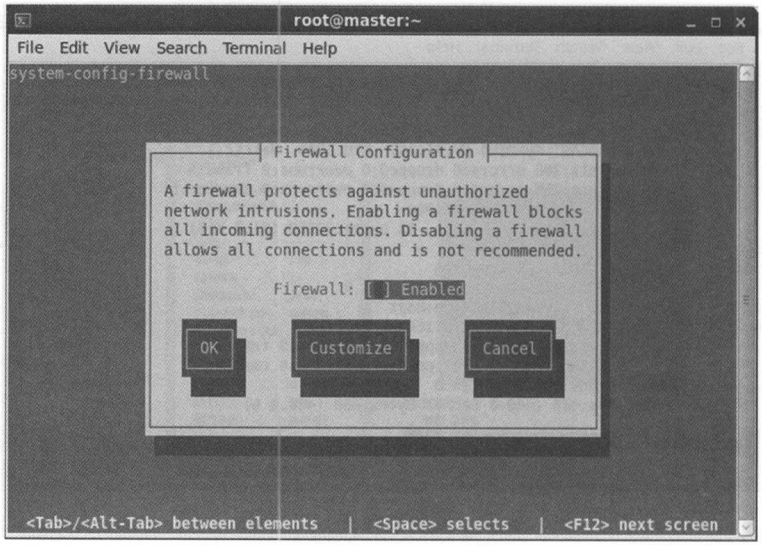


图 2.30　关闭防火墙示意图（2）

如果 Enabled 选项前为[*]，则将方括号中的*改为空格代替，表示关闭防火墙，然后选择 OK 选项并保存。

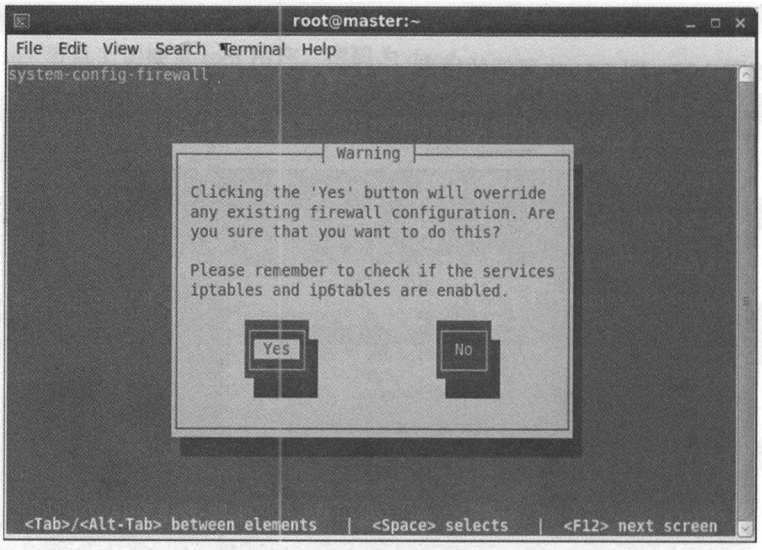


图 2.31　关闭防火墙示意图（3）

第 4 步，配置 hosts 文件。

分别在 master、slave1、slave2 节点的 root 用户下配置 hosts 文件。

```
$ gedit /etc/hosts
```

将以下代码添加到/etc/hosts 文件中：

```
192.168.149.132 master
192.168.149.133 slave1
192.168.149.134 slave2
```

其含义是 master 节点对应的 IP 地址为 192.168.149.132；slave1 节点对应的 IP 地址为 192.168.149.133；slave2 节点对应的 IP 地址为 192.168.149.134。

第 5 步，配置时钟同步。

在 master、slave1、slave2 节点的 root 用户下分别进行以下操作。

```
$ crontab -e
```

该命令是 vi 编辑命令，按键盘上 i 键进入编辑插入模式，输入下面代码：

```
0 1 * * * /usr/sbin/ntpdate cn.pool.ntp.org
```

按 Esc 键退出编辑，输入:wq 保存退出，然后通过执行下述命令手动同步时间。

```
$ /usr/sbin/ntpdate cn.pool.ntp.org
```

2.4.3　安装 JDK

分别在 master、slave1、slave2 节点的 root 用户下安装 JDK，每个节点上的步骤如下：

第 1 步，下载正确版本的 JDK。

本教材中演示需要 JDK1.7 及以上版本，读者可在 Oracal 官网上下载。

第 2 步，解压 JDK 文件，并生成相应目录。

首先创建/usr/java 目录，然后将 jdk-7u71-linux-x64.gz 移至该目录下并对其解压。

```
$ mv jdk-7u71-linux-x64.gz /usr/java/
$ cd /usr/java
$ tar -xvf jdk-7u71-linux-x64.gz
```

第 3 步，配置环境变量。

```
$ gedit /home/dong/.bash_profile
```

在.bash_profile 文件中配置以下配置以下环境变量：

```
export JAVA_HOME=/usr/java/jdk1.7.0_71/
export PATH=$JAVA_HOME/bin:$PATH
```

使得配置文件生效：

```
$ source /home/dong/.bash_profile
```

第 4 步，测试配置。

```
$ java -version
```

执行后如图 2.32 所示。

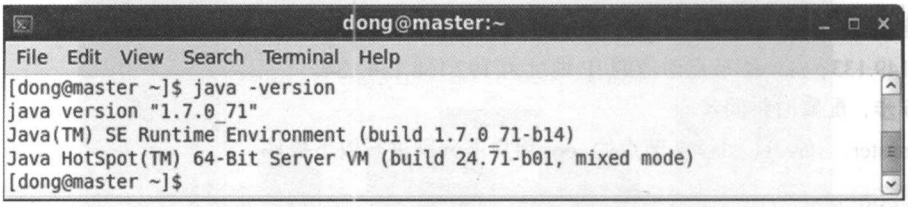


图 2.32　Java 安装示意图

2.4.4　免密钥登录配置

在 dong 用户下进行以下操作：

第 1 步　master 节点上通过下述命令生成密钥。

```
$ ssh-keygen -t rsa
```

该密钥文件在~/.ssh 中，共两个文件，如图 2.33 所示。

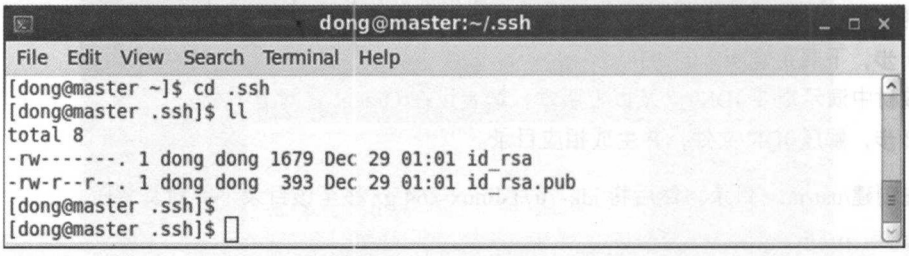


图 2.33　密钥目录示意图

第 2 步　复制公钥文件。

将其中的 id_rsa.pub 文件复制为 authorized_keys 文件，存放至~/.ssh 中。

```
$ cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
```

执行后如图 2.34 所示。

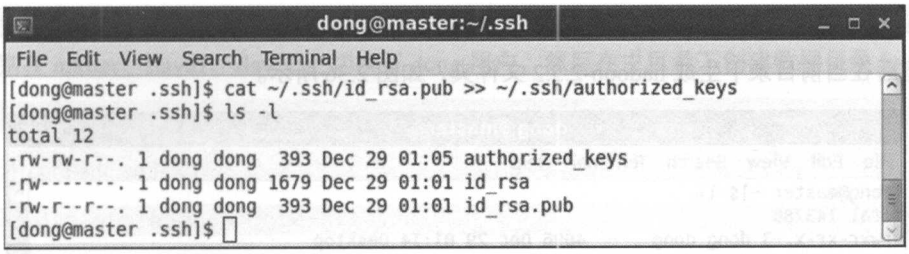


图 2.34　公钥目录示意图

第 3 步，修改 authorized_keys 文件的权限。

```
$ chmod 600 ~/.ssh/authorized_keys
```

第 4 步，将 authorized_keys 文件复制到 slave1、slave2 节点。

```
$ scp ~/.ssh/authorized_keys dong@slave1:~/
$ scp ~/.ssh/authorized_keys dong@slave2:~/
```

第 5 步，在 slave1、slave2 节点上执行以下命令，分别生成密钥。

```
$ ssh-keygen -t rsa
$ mv authorized_keys ~/.ssh/
$ cd ~/.ssh
$ chmod 600 authorized_keys
```

第 6 步，验证免密钥登录。

在 master 节点执行下述命令：

```
$ ssh slave1
```

如果出现图 2.35，说明免密钥登录成功。

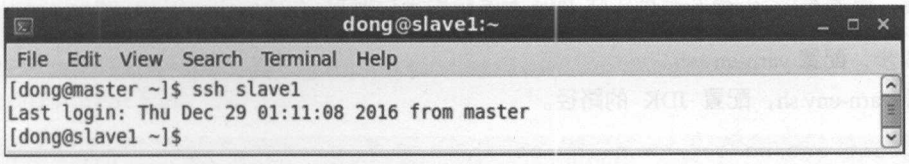


图 2.35　master 免密钥登录 slave1 示意图

类似地，在 master 节点可对 slave2 的免密钥登录进行验证。

2.4.5　Hadoop 配置

第 1 步，下载并解压安装包。

首先将压缩包移至 dong 用户的主目录下，然后执行下述命令对其进行解压。

```
$ tar -xvf hadoop-2.5.2.tar.gz
```

```
$ cd hadoop-2.5.2
```

解压后在当前目录下生成 hadoop-2.5.2 文件夹，如图 2.36 所示。

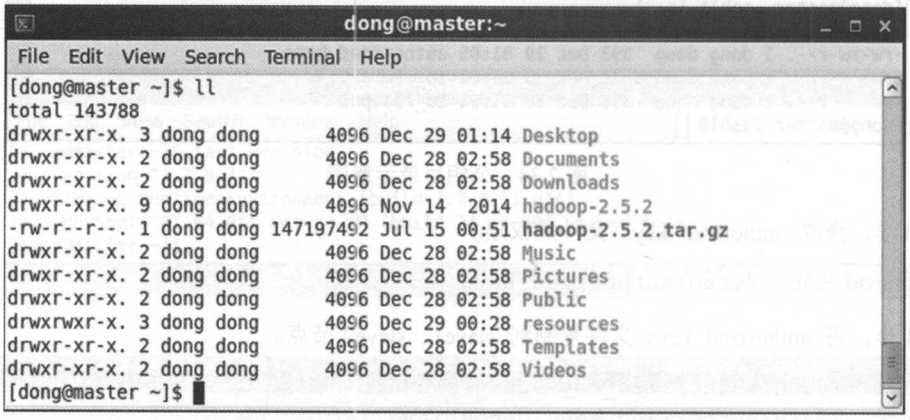


图 2.36　hadoop 安装包解压后目录示意图

第 2 步，配置 hadoop-env.sh。

打开 hadoop-env.sh，配置 JDK 的路径。

```
$ gedit /home/dong/hadoop-2.5.2/etc/hadoop/hadoop-env.sh
```

在文件靠前的部分找到下面的一行代码：

```
#export JAVA_HOME=${JAVA_HOME}
```

将这行代码修改为以下形式：

```
export JAVA_HOME=/usr/java/jdk1.7.0_71/
```

注意，需要将#号去掉并根据实际 Java 安装路径进行设置。

第 3 步，配置 yarn-env.sh。

打开 yarn-env.sh，配置 JDK 的路径。

```
$ gedit etc/hadoop/yarn-env.sh
```

在文件靠前的部分找到下面的一行代码：

```
# export JAVA_HOME=/home/y/libexec/jdk1.6.0/
```

将这行代码修改为以下形式：

```
export JAVA_HOME=/usr/java/jdk1.7.0_71/
```

注意，需要将#号去掉并根据实际 Java 安装路径进行设置。

第 4 步，配置 core-site.xml。

分别在 master、slave1、slave2 上切换至 dong 用户，然后在主目录下创建数据目录 hadoopdata。

```
$ mkdir /home/dong/hadoopdata
```

打开 core-site.xml 文件。

```
$ gedit etc/hadoop/core-site.xml
```

用以下代码覆盖 core-site.xml 文件中的内容。

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<!-- Put site-specific property overrides in this file. -->
<configuration>
<property>
<name>fs.defaultFS</name>
<value>hdfs://master:9000</value>
</property>
<property>
<name>hadoop.tmp.dir</name>
<value>/home/dong/hadoopdata</value>
</property>
</configuration>
```

第 5 步，配置 hdfs-site.xml。

打开 hdfs-site.xml 文件。

```
$ gedit etc/hadoop/hdfs-site.xml
```

用下面的代码替换 hdfs-site.xml 中的内容。

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<!-- Put site-specific property overrides in this file. -->
<configuration>
<property>
<name>dfs.replication</name>
<value>1</value>
</property>
</configuration>
```

第 6 步，配置 yarn-site.xml。

打开 yarn-site.xml 文件。

```
$ gedit etc/hadoop/yarn-site.xml
```

用下面代码覆盖 yarn-site.xml 中的内容。

```
<?xml version="1.0"?>
<configuration>
<property>
<name>yarn.nodemanager.aux-services</name>
<value>mapreduce_shuffle</value>
</property>
<property>
<name>yarn.resourcemanager.address</name>
<value>master:18040</value>
</property>
<property>
<name>yarn.resourcemanager.scheduler.address</name>
<value>master:18030</value>
</property>
<property>
<name>yarn.resourcemanager.resource-tracker.address</name>
<value>master:18025</value>
</property>
<property>
<name>yarn.resourcemanager.admin.address</name>
<value>master:18141</value>
</property>
<property>
<name>yarn.resourcemanager.webapp.address</name>
<value>master:18088</value>
</property>
</configuration>
```

第 7 步，配置 mapred-site.xml。

复制/home/dong/hadoop-2.5.2/etc/hadoop/目录下的 mapred-site.xml.template 文件，另存为该目录下的 mapred-site.xml。

```
$ cp etc/hadoop/mapred-site.xml.templateetc/hadoop/mapred-site.xml
```

用下面的代码覆盖 mapred-site.xml 中的内容。

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
<property>
<name>mapreduce.framework.name</name>
<value>yarn</value>
</property>
</configuration>
```

第 8 步，配置 slaves 文件。

打开 slaves 文件。

```
$ gedit etc/hadoop/slaves
```

用下面的代码替换 slaves 中的内容。

```
slave1
slave2
```

需要注意，此处既可以给出两个节点的 IP 地址，也可以给出机器名，但是要求每一个 IP 地址或机器名独占一行。

2.4.6　配置从节点

将 master 节点 dong 用户主目录下的 hadoop-2.5.2 文件夹分别复制至每个节点的 dong 用户主目录下。

```
$ scp -r hadoop-2.5.2 slave1:~/
$ scp -r hadoop-2.5.2 slave2:~/
```

2.4.7　配置系统文件

分别在 master、slave1、slave2 上以 dong 用户执行以下步骤。

第 1 步，配置系统文件。

```
$ gedit ~/.bash_profile
```

第 2 步，在打开的.bash_profile 文件中配置以下环境变量：

```
export HADOOP_HOME=/home/dong/hadoop-2.5.2
export PATH=$HADOOP_HOME/bin:$HADOOP_HOME/sbin:$PATH
```

第 3 步，使其配置生效。

```
$ source .bash_profile
```

2.4.8　启动 Hadoop 集群

第 1 步，格式化文件系统。

在 master 节点上执行下面的格式化命令。

```
$ hdfs namenode -format
```

第 2 步，启动 Hadoop。

进入 hadoop-2.5.2 文件夹。

```
$ cd ~/hadoop-2.5.2
```

执行启动命令。

```
$ sbin/start-all.sh
```

执行命令后，提示输入 yes/no 时，输入 yes。

第 3 步，验证。

master 的终端窗口执行 jps 指令后会出现 4 个进程，如图 2.37 所示；在 slave1、slave2 的终端窗口执行 jps 指令后会出现 3 个进程，分别如图 2.38 和图 2.39 所示。

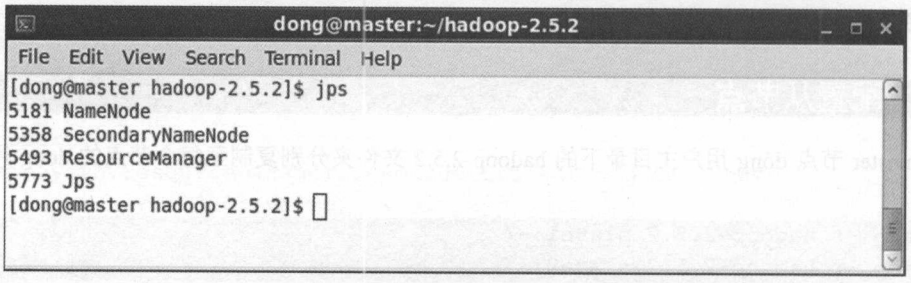


图 2.37　master 节点进程示意图

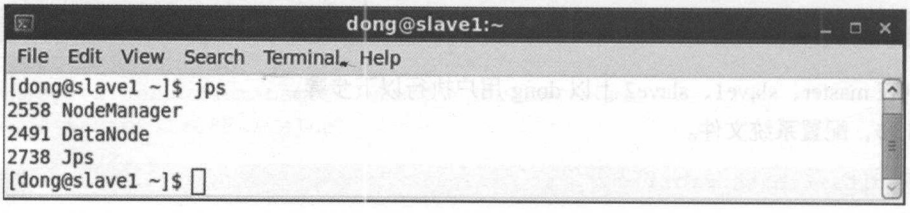


图 2.38　slave1 节点进程示意图

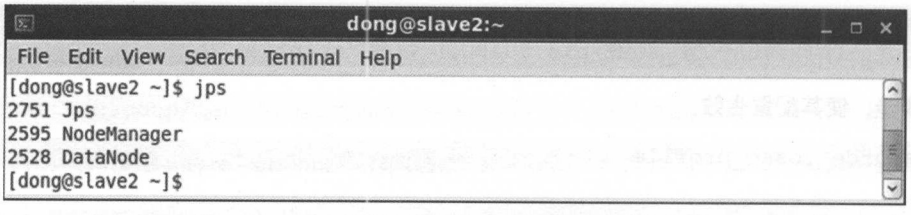


图 2.39　slave2 节点进程示意图

同时也可以通过 Web UI 查看集群是否成功启动。在 master 上启动 Firefox 浏览器，在浏览器地址栏中输入 http://master:50070/，检查 NameNode 和 DataNode 是否正常启动。图 2.40 表示 hadoop 正常启动。

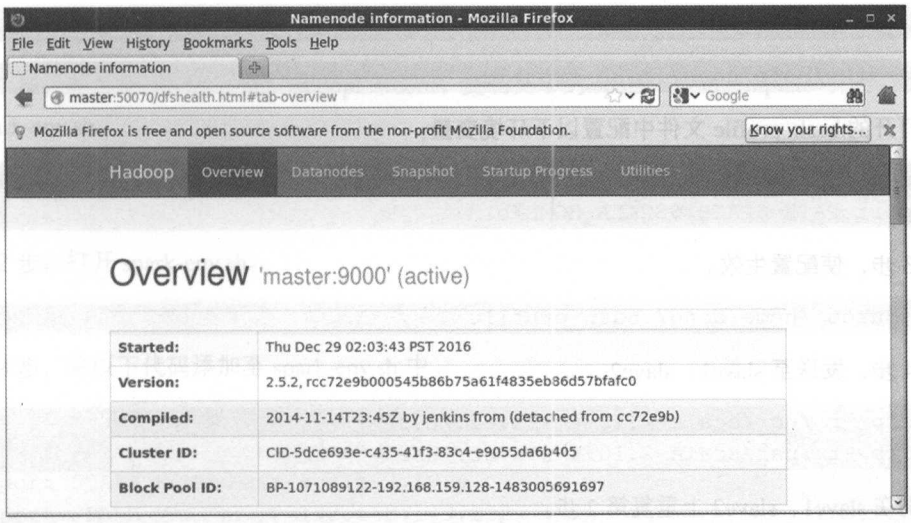


图 2.40　Web UI 查看 hadoop 示意图

在 master 启动 Firefox 浏览器，在地址栏中输入 http://master:18088/，检查 Yarn 是否正常启动。如图 2.41 所示 Yarn 正常启动。

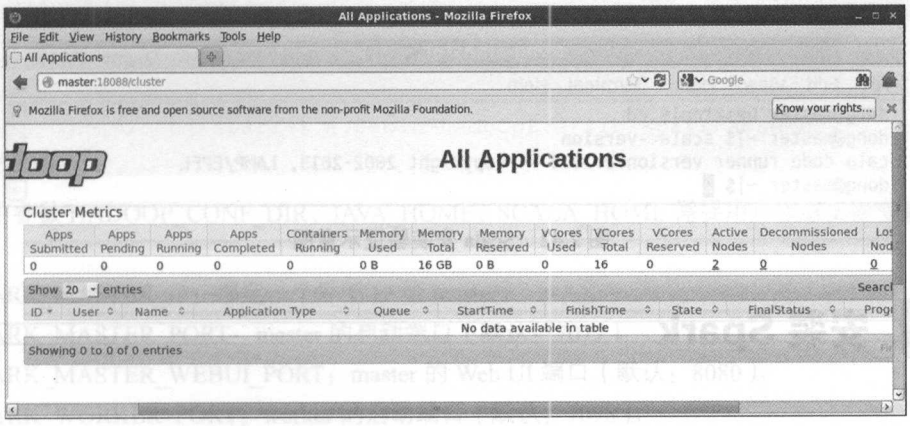


图 2.41　Web UI 查看 yarn 示意图

2.5　安装 Scala

第 1 步，下载并解压 scala 压缩包。

在 http://www.scala-lang.org 下载 scala 压缩包并移至 dong 用户的主目录下，然后解压。

```
$ tar -zvxf scala-2.10-2.10.4.tgz
```

第 2 步，配置系统文件。

```
$ gedit /home/dong/.bash_profile
```

在打开的.bash_profile 文件中配置以下环境变量：

```
export SCALA_HOME=/home/dong/scala-2.10.4
export PATH=$PATH:$SCALA_HOME/bin
```

第 3 步，使配置生效。

```
$ source /home/dong/.bash_profile
```

第 4 步，发送至 slave1、slave2。

```
$ scp -r /usr/scala-2.10.4 slave1:/usr
$ scp -r /usr/scala-2.10.4 slave2:/usr
```

分别在 slave1、slave2 上重复第 2 步。

第 5 步，验证。

分别在 master、slave1、slave2 执行下述语句验证，如果安装成功会显示如图 2.42 中所示的版本信息。

```
$ scala -version
```

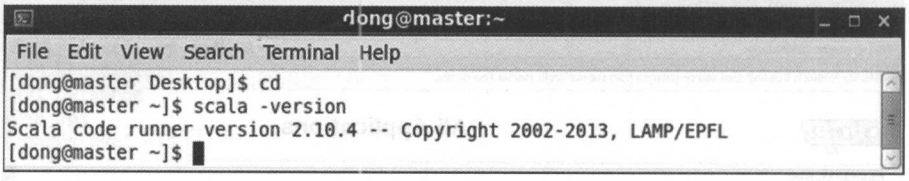


图 2.42　Scala 安装验证示意图

2.6 安装 Spark

2.6.1 下载并解压 Spark 安装包

在 Apache 官网下载相应版本的 Spark，然后移至 master 节点 dong 用户主目录下，并对其解压。

```
$ tar -zvxf spark-1.4.0-bin-hadoop2.4.tgz
```

执行该命令后会在 dong 的主目录下生成 spark-1.4.0-bin-hadoop2.4 文件夹。

2.6.2　配置 Spark-env.sh

第 1 步，进入 spark-1.4.0-bin-hadoop2.4/conf，复制其中的 spark-env.sh.template 并另存为该目录下的 spark-env.sh。

```
$ cd spark-1.4.0-bin-hadoop2.4/conf
$ cp spark-env.sh.template spark-env.sh
```

第 2 步，打开 spark-env.sh。

```
$ gedit spark-env.sh
```

第 3 步，将以下代码添加至 spark-env.sh 中。

```
export HADOOP_CONF_DIR=/home/dong/hadoop-2.5.2/
export JAVA_HOME=/usr/java/jdk1.7.0_71/
export SCALA_HOME=/usr/scala-2.10.4
export SPARK_MASTER_IP=192.168.149.132
export SPARK_MASTER_PORT=7077
export SPARK_MASTER_WEBUI_PORT=8080
export SPARK_WORKER_PORT=7078
export SPARK_WORKER_WEBUI_PORT=8081
export SPARK_WORKER_CORES=1
export SPARK_WORKER_INSTANCES=1
export SPARK_WORKER_MEMORY=2g
export
SPARK_JAR=/home/dong/spark-1.4.0-bin-hadoop2.4/lib/spark-assembly-1.4.0-hado
op2.4.0.jar
```

其中参数 HADOOP_CONF_DIR、JAVA_HOME、SCALA_HOME 需要用户根据实际安装情况进行配置。

SPARK_MASTER_IP：绑定一个外部 IP 给 master。

SPARK_MASTER_PORT：master 的启动端口（默认：7077）。

SPARK_MASTER_WEBUI_PORT：master 的 Web UI 端口（默认：8080）。

SPARK_WORKER_PORT：worker 的启动端口（默认：7078）。

SPARK_WORKER_CORES：作业可用的 CPU 内核数量（默认：所有可用）。

SPARK_WORKER_MEMORY：作业可使用的内存容量（默认：所有 RAM 去掉操作系统所用的 1 GB）。需要注意，该参数值必须是整数值，单位只能是 g 或 m，例如 2g 或者 1000m。

SPARK_WORKER_WEBUI_PORT：worker 的 Web UI 端口（默认：8081）。

2.6.3　配置 Spark-defaults.conf

第 1 步，进入 spark-1.4.0-bin-hadoop2.4/conf，复制 spark-defaults.conf.template 并另存为该目录

下的 spark-defaults.conf。

```
$ cd spark-1.4.0-bin-hadoop2.4/conf
$ cp spark-defaults.conf.template spark-defaults.conf
```

第 2 步，打开 spark-defaults.conf。

```
$ gedit spark-defaults.conf
```

第 3 步，向 spark-defaults.conf 添加以下代码：

```
spark.master=spark://192.163.149.132:7077
```

该参数需要根据用户 master 节点的实际 IP 地址进行配置。

2.6.4 配置 Slaves

第 1 步，进入 spark-1.4.0-bin-hadoop2.4/conf，复制其中的 slaves.template 并另存为该目录下的 slaves。

```
$ cp slaves.template slaves
```

第 2 步，打开 slaves。

```
$ gedit slaves
```

第 3 步，在 slaves 中添加子节点机器名或 IP 地址。

```
192.168.149.133
192.168.149.134
```

需要注意，在编辑 slaves 文件时每个机器名或 IP 地址独占一行。

2.6.5 配置环境变量

第 1 步，配置系统文件。

```
$ gedit /home/dong/.bash_profile
```

在打开的.bash_profile 文件中配置以下环境变量：

```
export SPARK_HOME=/home/dong/spark-1.4.0-bin-hadoop2.4/
export PATH=$PATH:$SPARK_HOME/bin
```

第 2 步，使配置生效。

```
$ source /home/dong/.bash_profile
```

2.6.6　发送至 Slave1、Slave2

```
$ scp -r ~/spark-1.4.0-bin-hadoop2.4 slave1:~/
$ scp -r ~/spark-1.4.0-bin-hadoop2.4 slave2:~/
```

分别在每个节点上重复 2.6.5 节中的步骤。

2.7　启动 Spark

第 1 步，启动 Hadoop。

第 2 步，进入 spark-1.4.0-bin-hadoop2.4 启动集群。

```
$ cd spark-1.4.0-bin-hadoop2.4
$ sbin/start-master.sh
$ sbin/start-slaves.sh
```

第 3 步，验证集群是否搭建成功，分别在 master、slave1、slave2 上通过 jps 命令查看进程，如图 2-43 所示。

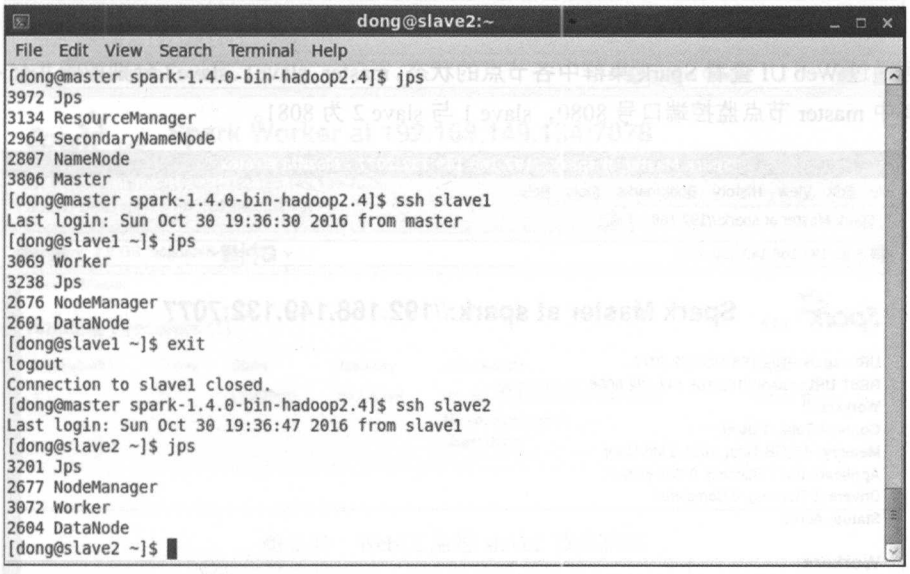


图 2.43　Spark 集群中各节点进程示意图

也可通过 Spark-shell 进一步验证 Spark 集群的安装情况。进入 spark-1.4.0-bin-hadoop2.4 执行如下指令：

```
$ bin/spark-shell
```

如果集群正常启动，Spark-shell 如图 2.44 所示。

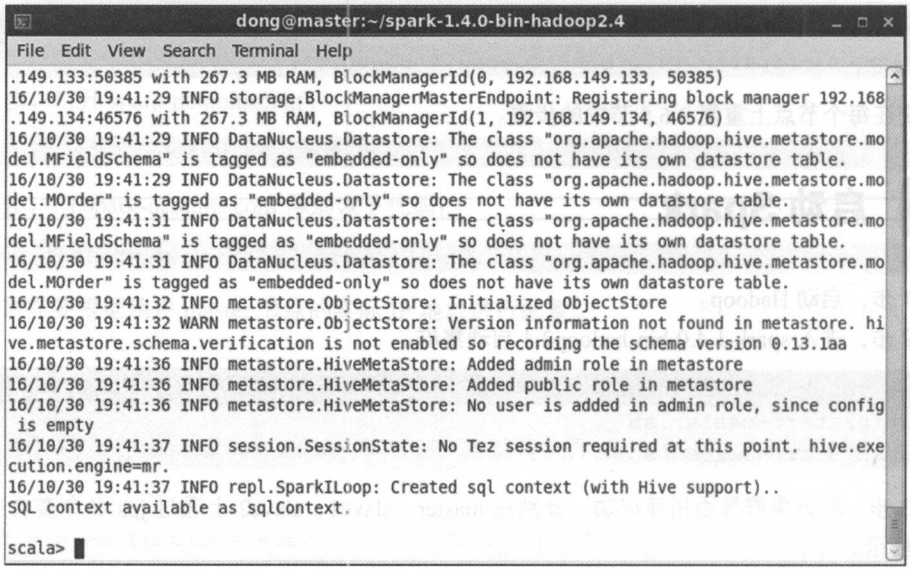


图 2.44　Spark-shell 交互窗口

也可通过 Web UI 查看 Spark 集群中各节点的状态，master、slave1、slave2 分别如图 2.45 ~ 图 2.47 所示。其中 master 节点监控端口号 8080，slave 1 与 slave 2 为 8081。

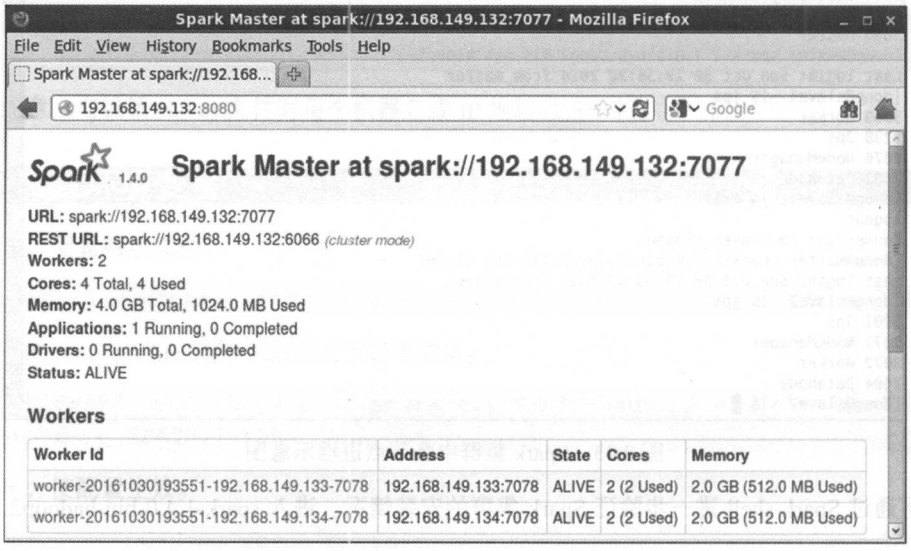


图 2.45　WebUI 监控 master 节点状态

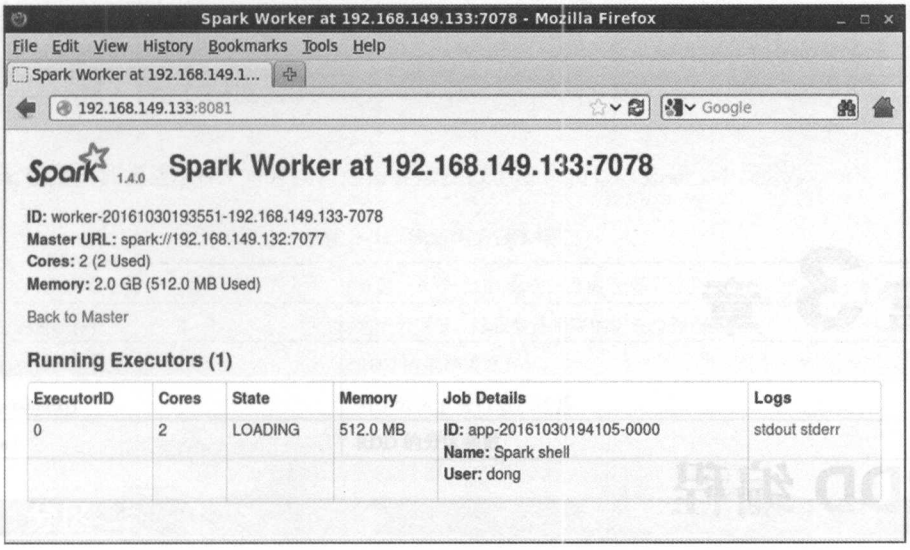


图 2.46　WebUI 监控 slave1 节点状态

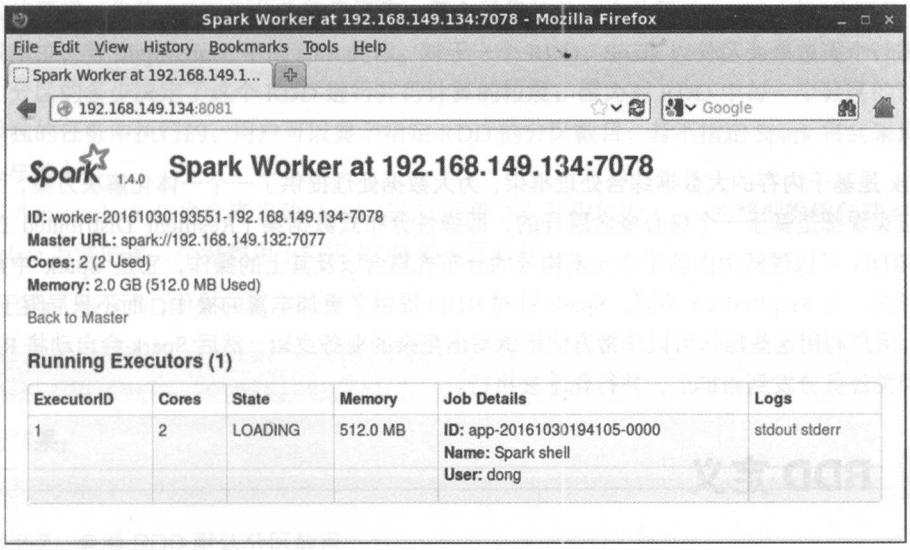


图 2.47　WebUI 监控 slave2 节点状态

至此，Standalone 模式的 Spark 集群搭建完成。

第3章

RDD 编程

Spark 是基于内存的大数据综合处理框架，为大数据处理提供了一个一体化解决方案，而该方案的设计与实现都是基于一个核心概念展开的，即弹性分布式数据集（Resilient Distributed Dataset，RDD）。RDD 可以理解为由若干个元素构成的分布式集合以及其上的操作，它是 Spark 中数据的重要组织形式。与 MapReduce 不同，Spark 针对 RDD 提供了更加丰富的操作，而不只局限于 Map 和 Reduce，用户利用这些操作可以非常方便地编写出复杂的业务逻辑，然后 Spark 会自动将 RDD 中的数据与相关任务分发到集群上，并行化地去执行。

3.1 RDD 定义

Apache 将 RDD 定义为弹性分布式数据集，它是 Spark 应用程序中数据的基本组织形式。弹性意味着 RDD 能够自动地进行内存和磁盘数据存储的切换，并且具有非常高的容错性；分布式说明 RDD 是一个存储在多个节点上的海量数据集合。RDD 是一种高度受限的共享内存模型，即 RDD 是只读的记录分区的集合。RDD 具有自动容错、位置感知调度和可伸缩性等数据流模型的特点。

3.2 RDD 的特性

Spark 在定义和描述 RDD 的时候，通常会涉及以下五个接口，如表 3-1 所示。

表 3-1　Spark RDD 接口

partition	分区，一个 RDD 会有一个或者多个分区
preferredLocations(p)	对于分区 p，返回数据所在的优先位置信息
dependencies()	RDD 的依赖关系
compute(p.context)	对于分区 p 进行迭代计算
partitioner()	RDD 的分区函数

3.2.1 分区

RDD 中的数据可能是 TB、PB 级别的，完全基于一个节点上的存储与计算并不现实。Spark 基于分布式的思想，先将 RDD 划分为若干子集，每个子集称为一个分区（partition）。分区是 RDD 的基本组成单位，与 MapReduce 中的 split 类似。对于一个 RDD，Spark 以分区为单位逐个计算分区中的元素。分区的多少决定了这个 RDD 进行并行计算的粒度，因为对 RDD 中每一个分区的计算都是在一个单独的任务中执行的。用户可以显示指定 RDD 的分区数目，若不指定 Spark 将会采用默认值（即 CPU 核数）。

通过./bin/spark-shell 命令来启动 Spark-shell. 示例 2-1 利用包含 1 ~ 5 的整型数组创建 RDD，其中 parallelize 方法的第二个参数用于指定 RDD 的分区数目。

示例 2-1　指定 RDD 分区数目为 4

```
scala>val rdd=sc.parallelize(1 to 5, 4)
scala>rdd.partitions.size
```

运行结果：

```
res0: Int=4
```

示例 2-2　查看 RDD 默认分区数目

```
scala>val rdd=sc.parallelize(1 to 5)
scala>rdd.partitions.size
```

运行结果：

```
res0: Int=2
```

3.2.2 依赖

RDD 是易转换、易操作的，这意味着用户可以从已有的 RDD 转换出新的 RDD。新、旧 RDD 之间必定存在着某种联系，这种联系称为 RDD 的依赖关系。RDD 间的依赖关系是 Spark 中的一个重要概念，是 Spark 进行容错、优化与任务调度的基础。

RDD 的依赖关系分为两种，如图 3.1 所示。

窄依赖：父 RDD 的每个分区最多被其子 RDD 的一个分区所依赖，也就是说子 RDD 的每个分区依赖于常数个父分区，子 RDD 每个分区的生成与父 RDD 的数据规模无关。

宽依赖：父 RDD 的每个分区被其子 RDD 的多个分区所依赖，子 RDD 每个分区的生成与父 RDD 的数据规模相关。

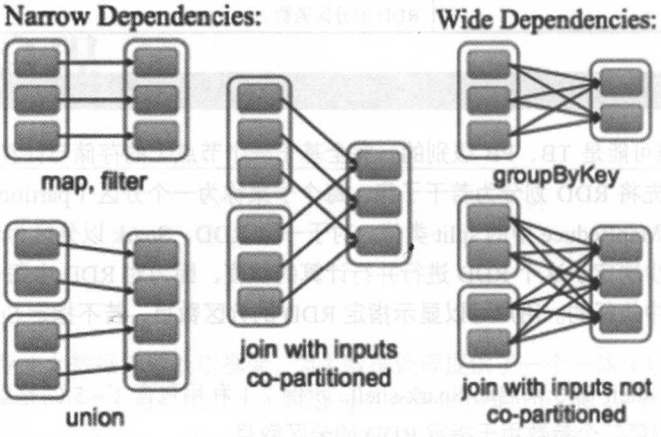


图 3.1 依赖关系

在图 3.1 中，透明的矩形框代表一个 RDD，每个矩形框里面的实心矩形代表 RDD 的一个分区。

为何要区分这两种依赖关系？一方面，对于若干个彼此窄依赖关系的 RDD，基于任何一个子 RDD 分区可以方便地计算出其所有祖先 RDD 的相应分区，这些 RDD 可以在集群的节点的内存中以流水线（pipeline）的方式高效执行，如图 3.2 中 Stage2 所示。另一方面，对于窄依赖关系间的 RDD，当子 RDD 的一个分区出错，可以很方便地利用父 RDD 中对应的分区重新计算该错误分区，因此窄依赖关系使得数据的容错与恢复非常方便；而对于宽依赖关系，子 RDD 的一个分区出错会导致其对应父 RDD 的多个分区进行重新计算，过程类似于 MapReduce 的 Shuffle 操作，代价非常高。

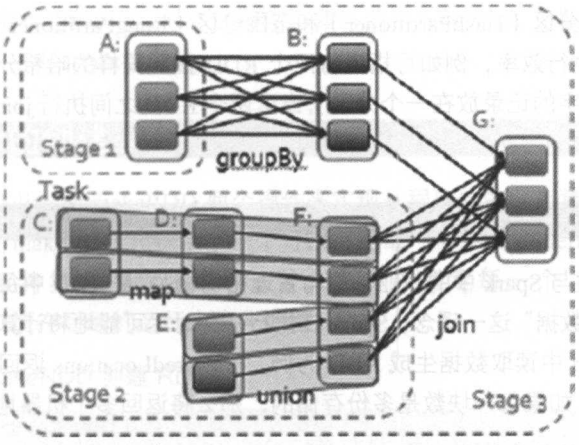


图 3.2 流水线形式计算多个窄依赖关系间的 RDD

示例 2-3 RDD 间的依赖关系

```
    scala>val rdd =sc.makeRDD(1 to 10)
    scala>val mapRDD=rdd.map(x=>(x, x))
    scala>mapRDD.dependencies
    scala>val                                shuffleRDD=mapRDD.partitionBy(new
org.apache.spark.HashPartitioner(3))
    scala>shuffleRDD.dependencies
```

运行结果：

```
    res1:
Seq[org.apache.spark.Dependency[_]]=List(org.apache.spark.OneToOneDependency
@45ec9c22)
    res2: Seq[org.apache.spark.Dependency[_]] = List(org.apache.spark.Shuffl
eDependency@7edbd4ec)
```

其中 res1 返回的 org.apache.spark.OneToOneDependency 为窄依赖关系；res2 返回的 org.apache.spark.ShuffleDependency 为宽依赖关系。

3.2.3 计算

Spark 中每个 RDD 中都包含一个函数，即在父 RDD 上执行何种计算后得到当前 RDD。每个 RDD 的计算都是以分区为单位的，而且每个 RDD 中的计算都是在对迭代器进行复合，不需要保存每次计算的结果。

3.2.4 分区函数

对于 Key-Value 形式的 RDD，Spark 允许用户根据关键字（Key）指定分区顺序，这是一个可选

的功能。目前支持哈希分区（HashPartitioner）和范围分区（RangPartitioner）。这一特性有助于提高 RDD 之间某些操作的执行效率，例如可指定将两个 RDD 按照同样的哈希分区方式进行分区（将同一机器上具有相同关键字的记录放在一个分区），当这两个 RDD 之间执行 join 操作时，会简化 Shuffle 过程，提高效率。

3.2.5 优先位置

RDD 优先位置属性与 Spark 中的作业调度与管理密切相关，是 RDD 中每个分区所存储的位置。遵循“移动计算不移动数据”这一理念，Spark 在执行任务时尽可能地将计算分配到相关数据块所在的节点上。以从 Hadoop 中读取数据生成 RDD 为例，preferredLocations 返回每一个数据块所在节点的机器名或者 IP 地址，如果每一块数是多份存储的，那么将返回多个机器地址。

示例 2-4　RDD 间的优先位置

```
    scala>val
rdd=sc.textFile("hdfs://master:9000/user/dong/input/file.txt").
    map(_.contains("l"))
    //根据依赖关系找到源头 rdd。
    scala>val hadoopRDD=rdd.dependencies(0).rdd
    //源头 rdd 的分区数。
    scala>hadoopRDD.partitions.size
    //查看第一个分区位置。
    scala>hadoopRDD.preferredLocations(HadoopRDD.partitions(0))
    //查看第二个分区位置。
    scala>hadoopRDD.preferredLocations(HadoopRDD.partitions(1))
```

运行结果：

```
    res0: Int = 2
    res1: Seq [String] = WrappedArray(slave2)
    res2: Seq [String] = WrappedArray(slave2)
```

运行结果表明 hadoopRDD 共有 2 个分区，均位于 slave2 上。

3.3 创建操作

RDD 中封装的操作非常丰富，可以大致分为转换操作（Transformation）和执行操作（Action）两类。在已知 RDD 上执行转换操作可返回一个新的 RDD；而执行操作则是向驱动器程序返回结果或把结果写入外部系统。转换操作采用了惰性策略，即转换操作并不会立即被计算，其相当于对业务逻辑的一种抽象描述，只有提交执行操作时，才会真正触发对转换操作的计算。本书按照功能进一步将 Spark 中的操作进行细分归类，本节将介绍创建 RDD 的相关操作。

得到一个RDD通常有三种方式：对驱动程序中集合的并行化处理、读取外部存储系统（HDFS、Hbase、Hive等）、基于已有RDD的转换。

3.3.1 基于集合的创建操作

Spark提供了parallelize和makeRDD两个操作来实现从程序中的已有集合创建RDD。这两个操作功能类似，不同的是makeRDD还提供了一个可以指定每一个分区preferredLocations参数的实现版本，makeRDD不仅将集合按顺序平均分片，还可以指定每一个RDD分区的优先位置(preferredLocations)，以便在后续的运行中优化调度。

示例3-1 利用makeRDD创建RDD

```
scala>val rdd=sc.makeRDD(1 to 10, 3)
scala>rdd.collect
```

运行结果：

```
res0: Array[Int] = Array(1,2,3,4,5,6,7,8,9,10)
```

示例3-2 利用parallelize创建RDD

```
scala>val rdd=sc.parallelize(1 to 10)
scala>rdd.collect
```

运行结果：

```
res1: Array[Int] = Array(1,2,3,4,5,6,7,8,9,10)
```

基于程序内部已有集合创建RDD是最简单直接的一种创建方式。用户可以在Shell中快速地创建出自己的RDD并对其进行操作，非常方便用户开发原型和测试。然而这种方式需要将整个数据集先存放在一台物理节点的内存中，因此在实际应用场景中并不多见，而通常是采用读取外部数据集的方式来创建RDD的。

3.3.2 基于外部存储的创建操作

Spark的整个生态系统与Hadoop是完全兼容的。Spark不仅可以将HDFS上的文件读取为分布式数据集，还支持其他所有实现了Hadoop接口的存储系统。Spark支持的Hadoop输入格式包括文本文件、SequenceFile、Avro、Parquet等。相应的Spark提供了textFile、HadoopFile, sequenceFile等一系列操作来读取相应类型的文件创建RDD。对于初学者，最常用的是textFile，即基于HDFS上的文本文件创建RDD。

示例3-3 textFile读取HDFS上指定文件创建RDD

```
scala>val rdd=sc.textFile("hdfs://master:9000/user/dong/input/file.txt")
scala>rdd.collect
```

读取 HDFS 上/user/dong/input 目录中的 file.txt 文件创建 RDD，file.txt 文件内容如下：

```
hello spark
hello hbase
spark spark
```

运行结果：

```
res0: Array[String] = Array(hello spark, hello hbase, spark spark)
```

不难发现，经过 textFile 后得到的是 MappedRDD，而不是 HadoopRDD。这是因为在执行 textFile 的过程中，还执行了其他的相关操作。可以用 toDebugString 这个命令来查看 RDD 的生成过程。

textFile 的应用方式非常灵活，不仅可以结合通配符使用，还可以读取压缩文件，例如示例 3-4~示例 3-6。

示例 3-4　textFile 读取 HDFS 上指定目录中所有文件创建 RDD

```
scala>val rdd =sc.textFile("hdfs://master:9000/user/dong/input/")
```

示例 3-5　借助通配符创建 RDD

```
scala>val rdd = sc.textFile("hdfs://master:9000/user/dong/input/*.txt")
```

示例 3-6　textFile 读取 HDFS 上指定目录中所有扩展名为.gz 的文件创建 RDD

```
scala>val rdd = sc.textFile("hdfs://master:9000/user/dong/input/*.gz")
```

需要注意，textFile 只支持.gz 格式的压缩文件。

textFile 带有一个可选的分区参数。用户也可以通过该参数指定创建 RDD 的分区数，但不能使用少于 block 个数。默认情况下 Spark 为每个 HDFS 中的 block 创建一个分区。

示例 3-7　基于程序 3-3 创建分区数为 4 的 RDD

```
scala>val rdd=sc.textFile("hdfs://master:9000/user/dong/input/file.txt", 4)
scala>rdd.collect
scala>rdd.partitions.size
```

运行结果：

```
res1: Int = 4
```

需要注意，textFile 同样可以读取本地磁盘上的数据，此时需要确保集群中每个节点上都要含有 textFile 中参数所指定的目录和文件。

对于创建 RDD，除了上述介绍的基于集合和基于外部存储两种方式外，还可以对已有 RDD 进行转换创建新 RDD。

3.4 常见执行操作

执行操作（action）是向应用程序返回值或向存储系统导出数据的操作，常见的有 first、count、collect、take 等。

示例 3-8 执行操作演示

```
scala>val rdd=sc.makeRDD(1 to 10)
//返回 RDD 中第 1 个元素
scala>rdd.first
//统计 RDD 中元素个数
scala>rdd.count
//对 RDD 中元素累加求和
scala>rdd.reduce(_+_)
//对 RDD 中所有元素求积
scala>rdd.reduce(_*_)
//打印出 RDD 中所有元素
scala>rdd.collect
//返回集合中的前 3 个元素
scala>rdd.take(3)
//排序后返回前 3 个元素
scala>rdd.top(3)
```

运行结果：

```
res0: Int = 1
res1: Long = 10
res2: Int = 55
res3: Int = 3628800
res4: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
res5: Array[Int] = Array(1, 2, 3)
res6: Array[Int] = Array(10, 9, 8)
```

3.5 常见转换操作

转换操作能够从已有的 RDD 得到新的 RDD。Spark 提供了丰富的转换操作，本节将重点介绍常用的几种。

3.5.1 一元转换操作

1. map

map 操作针对 RDD 中的每个元素，进行指定的函数运算，转换成新的元素，所有新的元素构成新的 RDD。

示例 3-8 map 操作演示

```
scala>val rdd1 = sc.parallelize(1 to 9, 3)
scala>val rdd2 = rdd1.map(x => x*2)
scala>rdd1.collect
scala>rdd2.collect
```

运行结果：

```
res0: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9)
res1: Array[Int] = Array(2, 4, 6, 8, 10, 12, 14, 16, 18)
```

示例 3-8 首先创建 rdd1，其中含有整数 1~9；对 rdd1 中的每个元素执行乘 2 操作，得到新的元素，所有新元素构成 rdd2；最后输出 rdd1 与 rdd2。

2. mapValues

mapValues 适用于 Key-Value 对形式的 RDD，即 RDD 中的每个数据都是有序二元组。针对 Key-Value 对中的每个 Value 值进行指定的函数运算（一对一映射），得到新的 Value 值；原 RDD 中的 Key 保持不变，并与新的 Value 构成新的 Key-Value 对，所有新的 Key-Value 对构成新的 RDD。

示例 3-9 mapValues 操作演示

```
scala>val rdd1 = sc.parallelize(List("dog", "tiger", "lion", "cat",
"panther", "eagle"))
scala>val rdd2 = rdd1.map(x => (x.length, x)).collect
scala>rdd2.mapValues("x" + _ + "x").collect
```

运行结果：

```
res0: Array[(Int, String)] = Array((3,xdogx), (5,xtigerx), (4,xlionx),
(3,xcatx), (7,xpantherx), (5,xeaglex))
```

示例 3-9 首先以字符串列表作为数据源创建 rdd1；然后利用 map 操作将 rdd1 中的每个字符串转换为 Key-Value 对（有序二元组），其中 Key 值为字符串长度，Value 值为字符串本身，例如"dog"经运算后对应得到 (3, "dog")；再利用 mapValues 操作将 rdd1 中的每个 Key-Value 对中的 Value 值执行字符串连接操作，例如(3, "dog")中的"dog"经运算后得到"xdogx"，然后与其 Key 值 3 构成新的 Key-Value 对(3,xdogx)，进而得到 rdd2；最后输出 rdd2。

3. mapPartitions

mapPartitions是map的一个变形。map的输入函数是作用于RDD中每个元素，而mapPartitions的输入函数作用于每个分区。

示例3-10 mapValues操作演示

```
scala>val a = sc.parallelize(1 to 9, 3)
def myfunc[T](iter: Iterator[T]) : Iterator[(T, T)] = {
var res = List[(T, T)]()
var pre = iter.next
while (iter.hasNext) {
val cur = iter.next; res .::= (pre, cur); pre = cur; }
res.iterator }
scala>a.mapPartitions(myfunc).collect
```

运行结果：

```
res0: Array[(Int, Int)] = Array((1,2), (2,3), (4,5), (5,6) , (7,8), (8,9))
```

上述例子中的函数myfunc是把分区中一个元素和它的下一个元素组成一个Tuple。因为分区中最后一个元素没有下一个元素了，所以(3,4)和(6,7)不在结果中。

4. flatMap

flatMap与map类似，针对RDD中的每个元素，经过指定的函数（一对多映射）运算后生成若干个新元素，所有新元素构成新RDD。

示例3-11 flatMap操作演示

```
scala>val rdd1 = sc.parallelize(1 to 4, 2)
scala>rdd1.flatMap(x => 1 to x).collect
```

运行结果：

```
res0: Array[Int] = Array(1, 1, 2, 1, 2, 3, 1, 2, 3, 4)
```

示例3-11首先以整数1~4作为数据源创建rdd1；利用flatMap操作对rdd1中的每个元素x进行运算产生x个新元素（从1到x），例如整数3经运算后产生1、2、3共三个新元素；所有新元素构成rdd2。

5. flatMapValues

flatMapValues类似于mapValues，适用于Key-Value对形式的RDD，对于每个Key-Value对中的Value值进行指定的函数运算，但是不同之处在于每个Value值可能被映射为若干个新值（不再是一对一映射），然后这些新值再与原Key组成一系列新的Key-Value对，所有新的Key-Value对构成新RDD。

示例 3-12　flatMapValues 操作演示

```
scala>val rdd1 = sc.parallelize(List((1,2),(3,4)))
scala>val rdd2 = rdd1.flatMapValues(x=>x.to(5))
scala>rdd2.collect
```

运行结果：

```
res0: Array[(Int, Int)] = Array((1,2), (1,3), (1,4), (1,5), (3,4), (3,5))
```

示例 3-12 首先以整型二元组列表作为数据源创建 rdd1；利用 flatMapValue 操作对 rdd1 中每个 Key-Value 对中的 Value 值 x 进行运算，对应得到从 x 到 5 若干个整数，每个运算得到的整数再与 Key 值构成新的 Key-Value 对，例如 rdd1 中的(1,2)，Value 值 2 经运算后得到 4 个整数：2、3、4、5，然后再与原 Key 值 1 构成 4 个新的 Key-Value 对：(1,2)、(1,3)、(1,4)、(1,5)，所有新的 Key-Value 对构成 rdd2。

6. groupByKey()

groupByKey 针对 Key-Value 对形式的 RDD，以 Key 值作为依据，对 Value 值进行分组。

示例 3-13　groupByKey 操作演示

```
scala>val rdd = sc.parallelize(List(("A",2),("B",4),("C",6),("A",3),("C",7)))
scala>rdd.groupByKey().collect
```

运行结果：

```
res0: Array[(String, Iterable[Int])] = Array((A, CompactBuffer(2, 3)), (B, CompactBuffer(4)), (C, CompactBuffer(6,7)))
```

7. sortByKey

sortByKey 针对 Key-Value 对形式的 RDD，以 Key 值作为依据进行排序。

示例 3-14　sortByKey 操作演示

```
scala>val rdd= sc.parallelize(List(("A",2),("B",4),("C",6),("A",3),("C",7)))
scala>rdd.sortByKey().collect
```

运行结果：

```
res0: Array[(String, Int)] = Array((A, 3), (A, 2) , (B, 4) , (C, 7) , (C, 6))
```

需要注意，sortByKey 默认对 Key 值进行升序排序，不对 Value 值进行排序，若想实现降序排序，可将示例 3-14 中的第二条语句改写为：

```
scala>val sortRdd=rdd.sortByKey(false).collect
```

运行结果：

```
res0: Array[(String, Int)] = Array((C, 6), (C, 7), (B, 4) , (A, 2), (A, 3))
```

8. reduceByKey

reduceByKey 针对 Key-Value 形式的 RDD，对具有相同 Key 的 Value 进行指定的函数运算，再将计算结果与 Key 值组成一个新的 Key-Value 对，所有新的 Key-Value 对构成新的 RDD。

示例 3-14 reduceByKey 操作演示

```
scala>val rdd1 = sc.parallelize(List(("A",2), ("A",1), ("B",4),("B",6)))
scala>rdd1.reduceByKey((x,y) => x + y).collect
```

运行结果：

```
res0: Array[(String, Int)] = Array(("A",3), ("B",10))
```

示例 3-14 首先以(String, Int)二元组列表作为数据源创建 rdd1；利用 reduceByKey 操作对 rdd1 中的元素按照 Key 值进行分组，共得到两组：以"A"作为 Key 值的所有 Value 一组("A",(2, 1))，以"B"作为 Key 值的所有 Value 一组("B",(4, 6))；组内的元素进行累加操作，再将结果与 Key 值构成新的 Key-Value 对，例如，("B",(4, 6))对应得到("B",10)。需要注意，reduceByKey 中指定的函数运算必须是一个二元函数。

9. filter

filter 针对 RDD 中的每一个元素进行指定的函数运算，对于返回值为 true 的元素，筛选出来作为新 RDD 中的元素。需要注意，filter 中函数参数的返回值类型必须是布尔型。

示例 3-15 filter 操作演示

```
scala>val rdd=sc.makeRDD(1 to 10).filter(_%3==0)
scala>rdd.collect
```

运行结果：

```
res0: Array[Int] = Array(3, 6, 9)
```

示例 3-15 首先以整数 1~10 作为数据源创建 RDD；然后利用 filter 操作对 RDD 中的每个整数执行函数运算，即判断该整数模 3 的结果是否为 0，若返回值为 true，则将该整数筛选出来作为 rdd 中的元素。

3.5.2 二元转换操作

尽管 RDD 不是严格意义上的集合，但是同样支持许多数学上的集合运算。

1. union

union 操作是将两个 RDD 中的元素进行合并，将结果封装成新的 RDD，类似于两个集合的并运算。

示例 3-16　union 操作演示

```
scala>val rdd1 = sc.makeRDD (1 to 3, 1)
scala>val rdd2 = sc.makeRDD(2 to 4, 1)
scala>val unionRDD = rdd1.union (rdd2)
scala>unionRDD.collect
```

运行结果：

```
res0: Array[Int] = Array(1, 2, 3, 2, 3, 4)
```

由示例 3-16 运行结果可知，union 操作与数学中集合的并运算还是有区别的，其结果包含两个 RDD 中相同的元素，但不会去重叠。

2. intersection

intersection 操作接收另一个 RDD 作为参数，返回两个 RDD 中相同的元素并封装成新的 RDD，相当于两个集合的交运算。

示例 3-17　union 操作演示

```
scala>val rdd1 = sc.makeRDD (1 to 3, 1)
scala>val rdd2 = sc.makeRDD(2 to 4, 1)
scala>val intersectionRDD = rdd1.intersection(rdd2)
scala>intersectionRDD.collect
```

运行结果：

```
res0: Array[Int] = Array(2, 3)
```

3. subtract

subtract 操作接收另一个 RDD 作为参数，返回由只存在于第一个 RDD 中而不存在于第二个 RDD 中的所有元素组成的新 RDD，相当于两个集合的差运算。

示例 3-18　subtract 操作演示

```
scala>val rdd1 = sc.makeRDD (1 to 3, 1)
scala>val rdd2 = sc.makeRDD(2 to 4, 1)
scala>val subtractRDD = rdd1.subtract(rdd2)
scala>subtractRDD.collect
```

运行结果：

```
res0: Array [Int] = Array (1)
```

4. join

join 针对 Key-Value 对形式的两个 RDD 进行连接操作，连接的依据是 Key 值，其功能与关系型数据库中表的内连接操作相同。类似地，Spark 还提供了 leftOuterJoin、rightOuterJoin 等连接操作。

示例 3-19 join 操作演示

```
scala>val l=sc.parallelize(Array((1, 1), (1, 2), (2, 1), (3, 1)), 1)
scala>val r=sc.parallelize(Array((1, 'x'), (2, 'y'), (2, 'z'), (4, 'w')),
1)
scala>val joinrdd1=l.join(r).collect
```

运行结果：

```
joinrdd1: Array[(String, (Int, Char))] = Array((B, (1, y)), (B, (1, z)), (A,
(1, x)), (A, (2, x)))
```

示例 3-20 leftOuterJoin 操作演示

```
scala>val l=sc.parallelize(Array((1, 1), (1, 2), (2, 1), (3, 1)), 1)
scala>val r=sc.parallelize(Array((1, 'x'), (2, 'y'), (2, 'z'), (4, 'w')),
1)
scala>val joinrdd2=l.leftOuterJoin(r).collect
```

运行结果：

```
joinrdd2: Array[(String, (Int, Option(Char)))] = Array((B, (1, Some(y))),
(B, (1, Some(z))), (A, (1, Some(x))), (A, (2, Some(x))), (C, (1, None)))
```

示例 3-21 rightOuterJoin 操作演示

```
scala>val l=sc.parallelize(Array((1, 1), (1, 2), (2, 1), (3, 1)), 1)
scala>val r=sc.parallelize(Array((1, 'x'), (2, 'y'), (2, 'z'), (4, 'w')),
1)
scala>val joinrdd3=l.rightOuterJoin(r).collect
```

运行结果：

```
joinrdd3: Array[(String, (Option(Int), Char))] = Array((B, (Some(1), y)),
(B, (Some(1), z)), (A, (Some(1), x)), (A, (Some(2), x)), (D, (None, w)))
```

示例 3-19~示例 3-21 中首先创建了两个 Key-Value 对形式的 RDD：l 与 r；当 l 调用 join 操作与 r 进行连接时，只考虑二者中共同的 Key 值为连接依据；当 l 调用 leftOuterJoin 操作与 r 进行连接时，以 l 中的 Key 值作为连接依据，即返回结果中包含 l 的所有 Key 值；类似的，当 l 调用 rightOuterJoin 操作与 r 进行连接时，以 r 中的 Key 值作为连接依据，将返回结果中包含 r 的所有 Key 值。

3.6 持久化操作

默认情况下，对于每一个经过一系列转换操作后得到的 RDD，当在其上多次提交执行操作时，该 RDD 将会被重复计算。例如下面创建 rdd 的代码：

```
scala>val rdd=rdd1.map(func1).map(func2).map(func3).collect
scala>rdd1.count
```

在执行第一条语句时，rdd 已经通过 rdd1 和三个 map 操作创建完成；当执行第二条语句时，rdd 会被重复创建，即再次执行三个 map 操作。为了避免对一个 RDD 的重复计算生成，可以使用持久化操作。Spark 的持久化操作支持在内存、磁盘上保存数据集或在集群间复制数据集，以便后续查询同一 RDD 时能够快速访问。

Spark 中的持久化操作有 cache、persist、checkpiont。其中 persist 是将数据持久化到磁盘或内存；cache 可视为 persist 的特例，仅将数据持久化到内存；checkpoint 将数据持久化到磁盘上，与 persist 将数据持久化到硬盘上类似，但是 checkpoint 不再保存 RDD 间的依赖关系。

```
defpersist(newLevel: StorageLevel): RDD.this.type
```

persist 接收一个 StorageLevel 类型的对象 newLevel 作参数。查看 StorageLevel 类的源码，其通过 5 个成员字段封装了持久化的方式、位置等相关信息。

```
class StorageLevel private(
    private var _useDisk: Boolean,       //是否持久化到磁盘
    private var _useMemory: Boolean,     //是否持久化到内存
    private var _useOffHeap: Boolean,    //是否持久化到堆外内存
    private var _deserialized: Boolean, //是否反序列化
    private var _replication: Int = 1)  //持久化的副本数
  extends Externalizable
```

为了方便用户使用持久化操作，Spark 在 StorageLevel 类的伴生对象中定义了若干 StorageLevel 类型的成员字段，其中封装了不同的持久化信息。

```
object StorageLevel {
val NONE = new StorageLevel(false, false, false, false)
val DISK_ONLY = new StorageLevel(true, false, false, false)
val DISK_ONLY_2 = new StorageLevel(true, false, false, false, 2)
val MEMORY_ONLY = new StorageLevel(false, true, false, true)
val MEMORY_ONLY_2 = new StorageLevel(false, true, false, true, 2)
val MEMORY_ONLY_SER = new StorageLevel(false, true, false, false)
val MEMORY_ONLY_SER_2 = new StorageLevel(false, true, false, false, 2)
val MEMORY_AND_DISK = new StorageLevel(true, true, false, true)
val MEMORY_AND_DISK_2 = new StorageLevel(true, true, false, true, 2)
```

```
    val MEMORY_AND_DISK_SER = new StorageLevel(true, true, false, false)
    val OFF_HEAP = new StorageLevel(true, true, true, false, 1)
    val MEMORY_AND_DISK_SER_2=new StorageLevel(true, true, false, false, 2)
    ……
    }
```

cache()相当于 persist(MEMORY_ONLY)。

如果内存比较紧张就采用序列化形式存入内存，但是当进行反序列化数据的时候会消耗大量的CPU资源；反之，如果内存比较宽裕可以采用反序列化形式存入内存。

示例 3-22　cache 操作演示

```
    scala>val
rdd=sc.textFile("hdfs://master:9000/user/dong/input/sogou.txt")
    scala>rdd.cache
    scala>rdd.count
```

运行结果，如图 3.3 所示。

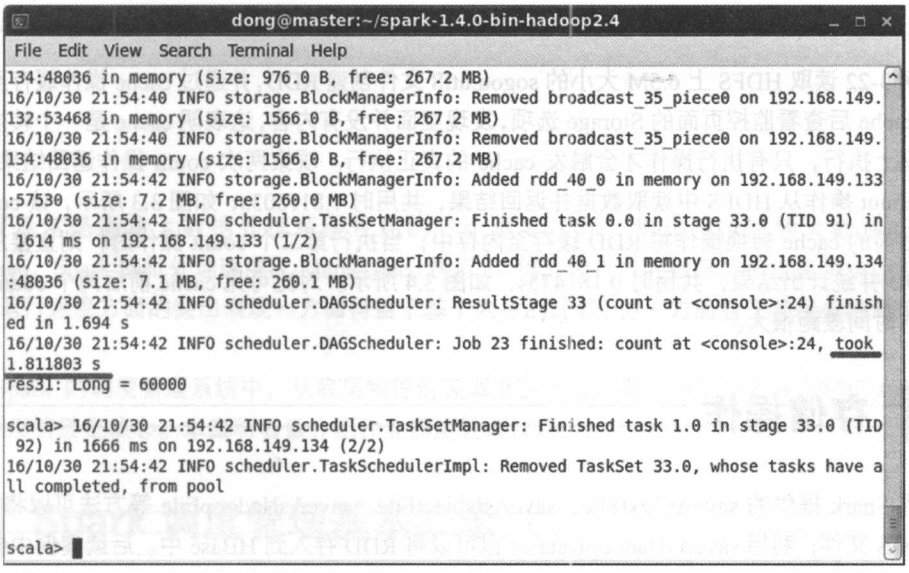


图 3.3　cache 操作演示图 1

```
    scala>rdd.count
```

运行结果，如图 3.4 所示。

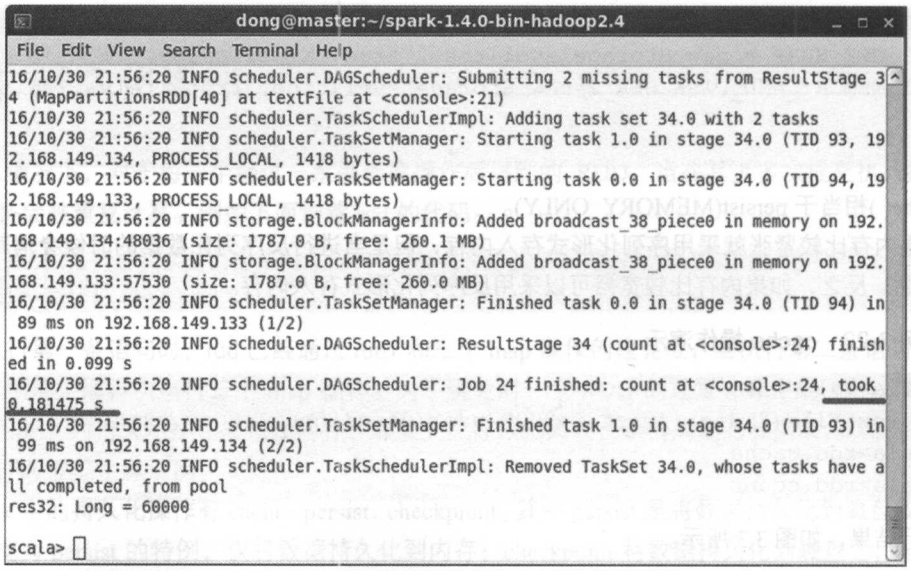


图 3.4　cache 操作演示图 2

示例 3-22 读取 HDFS 上 6.5M 大小的 sogou.utf8 文件创建 RDD，并通过 cache 操作缓存至内存。当执行 cache 后查看监控页面的 Storage 选项，发现里面并没有内容，这表明 cache 是一个转换操作，并不会马上执行，只有执行操作才会触发 cache 的真正执行。观察两次 count 操作返回结果的时间，第一个 count 操作从 HDFS 中读取数据并返回结果，共用时 1.811803s，如图 3.3 所示，该 count 操作触发了之前的 cache 转换操作将 RDD 缓存至内存中；当执行第二个 count 操作时，则直接从内存中读取 RDD 并统计出结果，共用时 0.181475s，如图 3.4 所示。对比可见 cache 前后两个 count 操作返回结果的时间差距很大。

3.7 存储操作

利用 Spark 提供的 saveAsTextFile、saveAsObjectFile、saveAsHadoopFile 等方法可以将 RDD 存储为 HDFS 文件；利用 saveAsHadoopDataset 也可以将 RDD 存入到 HBase 中。后续案例中会介绍相关方法的具体应用。

第4章

Spark 调度管理与应用程序开发

速度通常是衡量大数据处理系统性能的一个重要指标。相对于传统的大数据处理框架 MapReduce，Spark 的批处理速度比 MapReduce 快近 10 倍，内存中的数据分析速度则快近 100 倍。Spark 在运行速度方面的突出表现一方面得益于基于内存的计算，另一方面益于其优秀的调度管理策略。

在 Spark 的调度管理系统中，从底层物理资源调度到上层应用，涉及众多概念和相关模块，其中作业调度管理是核心。本章将着重介绍作业调度管理模块。

4.1 Spark 调度管理基本概念

1. Application

基于 Spark 的应用程序，由一个或多个作业组成。每个应用程序的执行涉及 Driver program 和集群上的 Executors 两部分。

2. Driver Program

DriverProgram 中包含了启动运行函数 main 函数和一个 SparkContext 类型的实例。其中

SparkContext 实例是 Application 与 Spark 集群进行交互的唯一通道，是每一个 Application 的入口。Driver Program 可以运行在任何可以提交作业的节点上（Worker，Master），并不是必须运行在 Master 节点上。在实际生产环境中为了减少 Master 的负担，尽量避免在 Master 上提交 Driver Program。

3. Worker

是 Spark 集群中任何可以运行 Application 相关代码的节点。

4. Executor

是 Worker 上为某个 Application 启动的一个进程，该进程负责执行任务并且负责将数据存储在内存或者磁盘中。Executor 拥有 CPU 和内存资源，它是资源管理系统能够给予的最小单位。一个 Worker 节点上可能有多个 Executor，但是每个 Executor 中仅执行一个 Application 对应的任务。也就是说，不同的 Application 通过 Executor 无法共享数据。

5. Task

Task 即任务，是被送到 Executor 上执行的工作单元。RDD 中的每个分区都对应相应的 Task，而每个 Task 对应于 Executor 中的一个线程，这使得系统更加轻量级，Task 之间切换的时间延迟更短。

6. Job

Job 即作业，是包含多个 Task 的并行计算，其与 Application 中的执行操作一一对应，也就是说 Application 每提交一个执行操作 Spark 对应生成一个 Job。

7. Stage

通常执行操作之前会有若干个转换操作，而每个转换操作意味着父 RDD 到子 RDD 的转换，因此一个 Job 中通常涉及多个 RDD。将 Job 中涉及的 RDD 进行分组，每组称为一个 Stage。

8. Cluster Manager

是在集群上获取资源的外部服务。若是 Standalone 模式的 Spark 集群，Cluster Manager 即为 Master；若是基于 Yarn 模式的 Spark 集群，Cluster Manager 即为 Resource Manager。

综合上述的基本概念，Spark Application 的执行框架如图 4.1 所示。一个 Spark Application 由两部分构成：Driver Program 和若干 Executor。其中 Driver Program 中包含了 Spark Context 和 main 函数；Executor 是 Worker 中用于运行 Task 的进程。在执行 Spark Application 过程中，集群上的资源如何调度，由 Cluster Manager 进行管理。

4.2 作业调度流程

每个作业从提交到完成都要经历多个步骤，如图 4.2 所示。一个 Spark 作业被逐步拆分成更小的调度单元，最终以任务的形式按照一定的逻辑依赖关系依次提交执行，并返回结果。

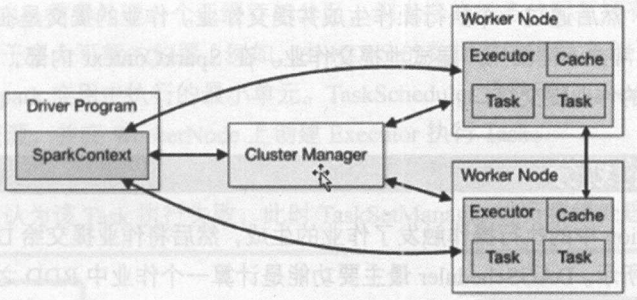


图 4.1　SparkApplication 的执行框架

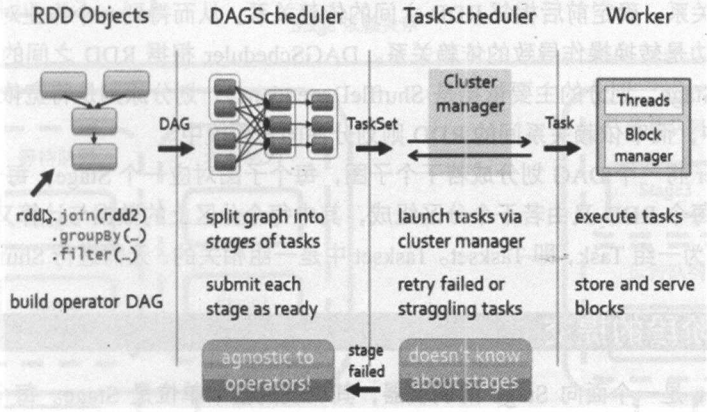


图 4.2　Spark 作业调度流程图

在 Spark 作业调度系统中，调度的前提是判断多个作业任务的依赖关系，这些作业任务之间可能存在因果的依赖关系，也就是说有些任务必须先获得执行，然后相关的依赖任务才能执行。

在作业调度的相关类中，最重要的就是 DAGScheduler 和 TaskScheduler。DAGScheduler 是基于 DAG 的调度类，负责将作业拆分成不同阶段的具有依赖关系的多批任务组，在逻辑层面实现对 Spark 作业的调度。TaskScheduler 则根据不同的任务需求，负责调度和管理每个任务所需的资源，在物理层面实现对 Spark 作业的执行。

4.2.1　作业的生成与提交

Spark Application 中的核心业务逻辑通常是针对若干 RDD 的一系列操作。操作可分为执行操作和转换操作两类，其中转换操作采用了惰性策略，即在实际编译和运行应用程序的过程中与转换操作相关的计算都是延迟执行的。转换操作仅仅生成对应的 RDD 依赖关系链，记录了 RDD 之间的业务逻辑。只有当应用程序提交了执行操作，才会触发之前一系列转换操作的真正计算，即触发 Spark 向集群提交实际作业。

如图 4.2 中 RDD Objects 部分所示，Spark Application 首先在粗描述粒度层面上明确每个子 RDD

所对应的父 RDD。然后通过一个执行操作生成并提交作业。作业的提交是在 RDD 的相关操作中隐式地被调用的，通常并不需要用户显式地提交作业。在 SparkContext 内部，作业的提交将进一步调用 DAGScheduler 的作业提交接口。

4.2.2 阶段的划分

Spark Application 中的执行操作触发了作业的生成，然后将作业提交给 DAGScheduler，如图 4.2 中 DAGScheduler 所示。DAGScheduler 最主要功能是计算一个作业中 RDD 之间的依赖关系，指定调度逻辑。DAGScheduler 从 RDD 依赖关系链末端的 RDD 出发，逆向遍历整个 RDD 依赖关系链，一方面明确了 Spark Application 中涉及哪些 RDD，另一方面从较细的粒度层面上分析父 RDD 分区与子 RDD 分区之间的关系，确定前后相邻 RDD 之间的依赖关系，从而得到一个作业对应的 DAG，图中的顶点是 RDD，边是转换操作导致的依赖关系。DAGScheduler 根据 RDD 之间的依赖关系将 DAG 图划分为若干个 Stage，划分的主要依据是 ShuffleDependency，划分原则是将宽依赖关系间的 RDD 划分在不同阶段中，而窄依赖关系间的 RDD 则划分在同一阶段中。

DAGScheduler 将一个 DAG 划分成若干个子图，每个子图对应一个 Stage，每个 Stage 中包含了若干个 RDD，而每个 RDD 又由若干个分区组成，其中每个分区上的数据与计算又对应着 Task。所以一个 Stage 可视为一组 Task，即 Taskset。Taskset 中是一组相关的、无需进行 Shuffle 操作的 Task。

4.2.3 调度阶段的提交

DAGScheduler 是一个面向 Stage 的调度器，其调度的基本单位是 Stage。每个 Stage 对应一个 Taskset，因此 DAGScheduler 是以 Taskset 的形式向 TaskScheduler 提交每个 Stage 的。

在 DAGScheduler 划分出的若干个 Stage 中，执行操作处理的 RDD 所在的 Stage 称为 FinalStage。DAGScheduler 进一步从这个 FinaIStage 生成一个作业实例，这两者的关系进一步存储在映射表中，用于在该调度阶段全部完成时做一些后续处理，如报告状态、清理作业相关数据等。

DAGScheduler 在具体提交一个 Stage 时，首先判断该 Stage 依赖的所有父 Stage 的结果是否可用，如果都可用，则提交当前 Stage；如果存在某个父 Stage 的结果不可用，则尝试再次提交不可用的父 Stage。在提交过程中，因依赖的 Stage 的结果不可用而没有提交成功的 Stage，都被放入一个等待队列中，等待被提交。提交流程如图 4.3 所示。

4.2.4 任务的提交与执行

DAGScheduler 将划分出的 Stage 以 Taskset 的形式提交给 TaskScheduler。一个 TaskScheduler 只为一个 SparkContext 服务。TaskScheduler 处理作业的粒度更加精细，它面向 Taskset 中的每一个 Task，以 Task 作为调度的基本单元。

提交后的每个 Taskset 会触发 TaskScheduler 构建一个 TaskSetManager 的实例来管理这个 Taskset 的生命周期。TaskSetManager 结合数据的存放位置，以最小化通信开销为原则，为每个 Task 选择运

行的节点。如果某个 Task 需要处理一个已缓存的分区，则直接将任务分配至拥有这个分区的节点；如果需要处理的分区位于多个可能的位置（例如，由 HDFS 的数据存放位置决定），则将任务分配给这一组节点。Task 是 Spark 应用中执行的最小单元。TaskScheduler 借助于 Cluster Manager 为每一个 Task 的执行申请系统资源，并在 WorkerNode 上创建 Executor 执行 Task。

此外，TaskSetManager 会追踪和监控每个 Task 的运行情况，如果一个 Task 被提交了 4 次（默认）都没有执行成功，则就认为该 Task 执行失败，此时 TaskSetManager 进行容错处理。

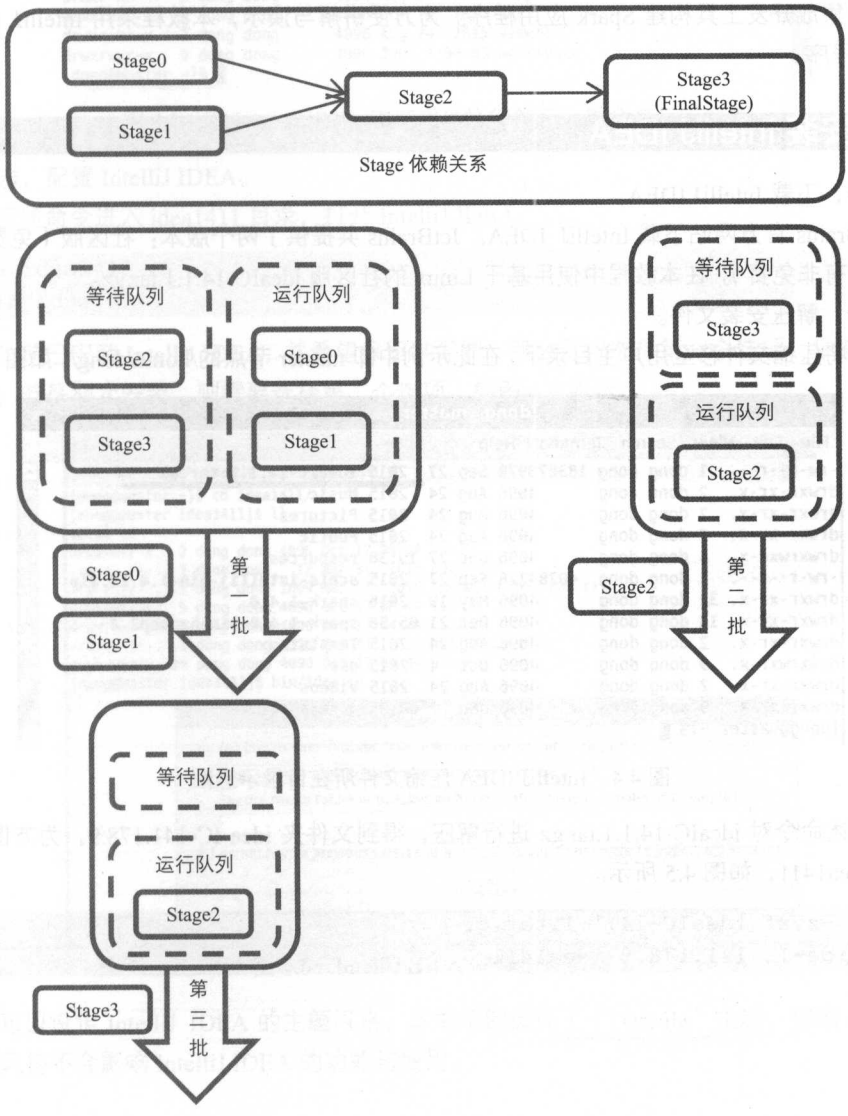


图 4.3　DAGScheduler 提交 Stage 的流程示意图

4.3 基于 IntelliJ IDEA 构建 Spark 应用程序

Spark Shell 是 Spark 自带的一个应用程序，其作为一个命令行交互工具，方便 Spark 初学者迅速掌握相关 API，也方便用户对简单的 Spark 应用程序进行开发与测试。然而在实际大数据工程项目开发过程中，面对的数据多是海量的、多源的、快速变的，数据集上的业务逻辑也是非常复杂的，因此需要使用集成开发工具构建 Spark 应用程序。为方便讲解与演示，本教程采用 IntelliJ IDEA 开发 Spark 应用程序。

4.3.1 安装 IntelliJ IDEA

第 1 步，下载 IntelliJ IDEA。

从 JetBrains 官方网站下载 IntelliJ IDEA。JetBrains 共提供了两个版本：社区版（免费开源）和终极版（专有非免费）。在本教程中使用基于 Linux 的社区版 ideaIC-14.1.1.tar.gz。

第 2 步，解压安装文件。

下载后将压缩文件移至用户主目录下，在此示例中即 master 节点的/home/dong/，如图 4.4 所示。

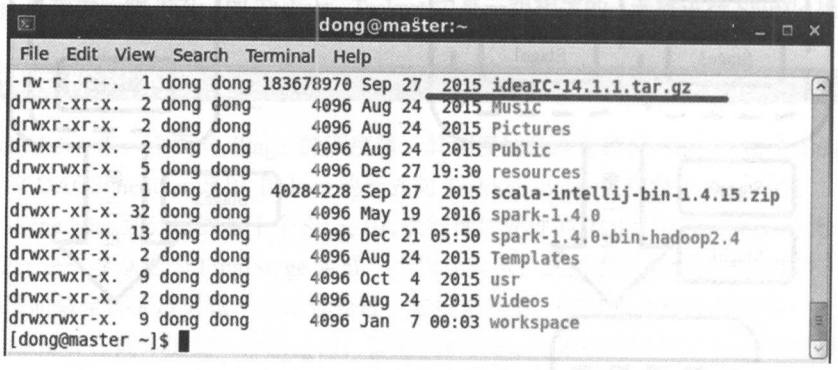


图 4.4 IntelliJ IDEA 压缩文件所在目录示意图

执行下述命令对 ideaIC-14.1.1.tar.gz 进行解压，得到文件夹 idea-IC-141.178.9，为方便操作将其重命名为 idea1411，如图 4.5 所示。

```
$ tar -zvxf ideaIC-14.1.1.tar.gz
$ mv ide-IC-141.178.9 idea1411
```

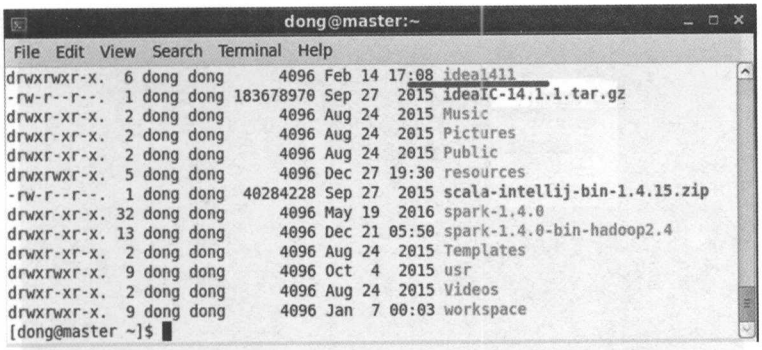


图 4.5　对 IntelliJ IDEA 安装文件执行解压、重命名后示意图

第 3 步，配置 IntelliJ IDEA。

通过下述命令进入 idea1411 目录，打开 IntelliJ IDEA。

```
$ cd idea1411
$ bin/idea.sh
```

如果先前已安装 IntelliJ IDEA，并希望保留其配置，那么可以选择第一个选项，并指定版本的配置文件夹；若是初次安装，则需要选择第二个选项，如图 4.6 所示。

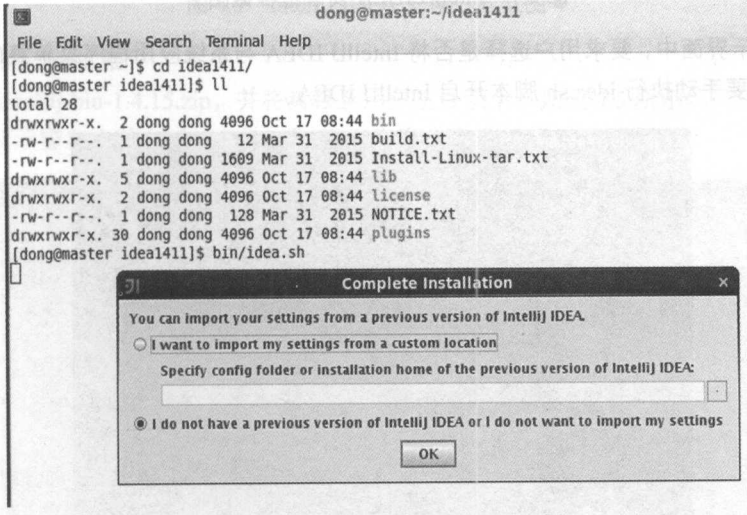


图 4.6　IntelliJ IDEA 安装过程截图 1

用户可以设定 IntelliJ IDEA 的主题风格，本书示例选择了“Darcula”主题，如图 4.7 所示。主题的不同风格不会影响 IntelliJ IDEA 的功能与使用。

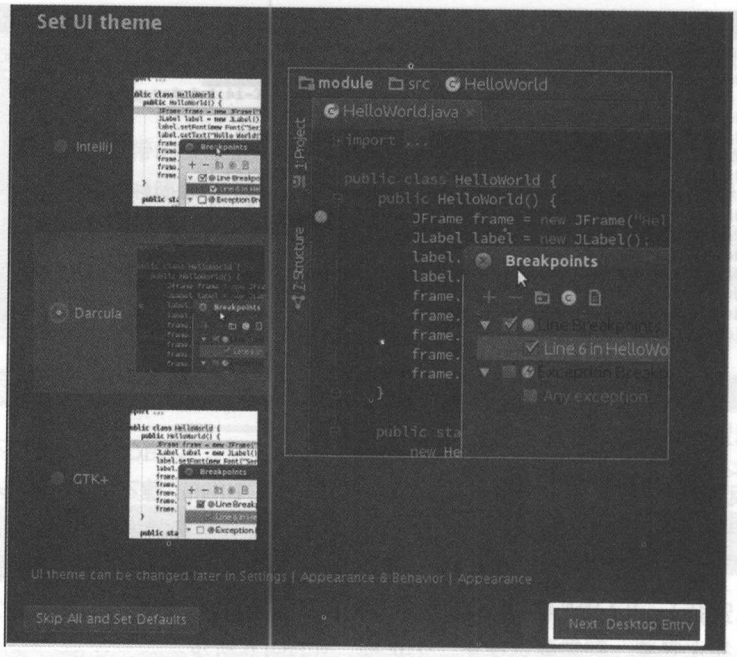


图 4.7　IntelliJ IDEA 安装过程截图 2

在图 4.8 所示界面中，要求用户选择是否将 IntelliJ IDEA 与系统应用程序菜单集成。若不勾选该选项，则用户需要手动执行 idea.sh 脚本开启 IntelliJ IDEA。

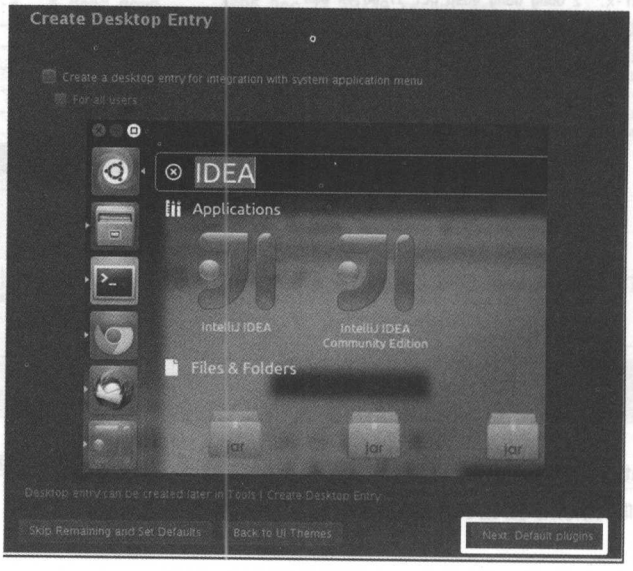


图 4.8　IntelliJ IDEA 安装过程截图 3

在图 4.9 所示的窗口中，用户可以选择所需的工具，包括构建工具、版本控制工具以及许多其他工具。

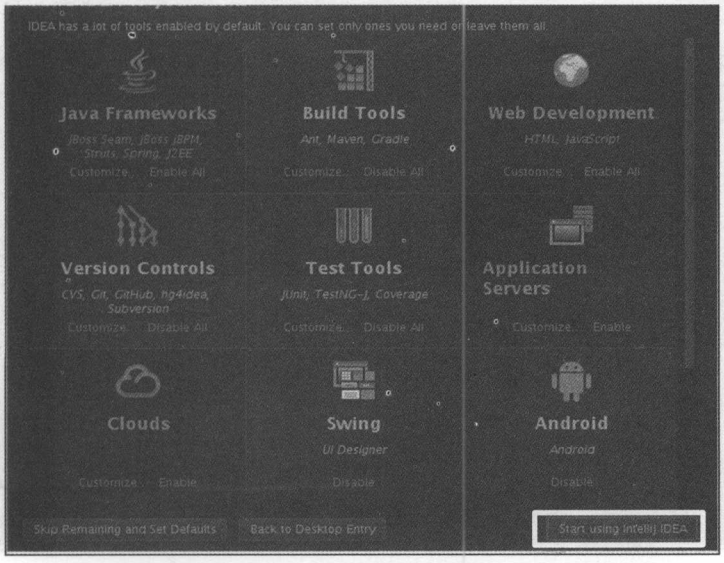


图 4.9　IntelliJ IDEA 安装过程截图 4

第 4 步，安装 Scala 插件。

下载 scala-intellij-bin-1.4.15.zip，并将其移至用户主目录下，如图 4.5 所示。

在图 4.10 所示窗口中单击“Configure”下拉菜单中的“Plugins”选项。

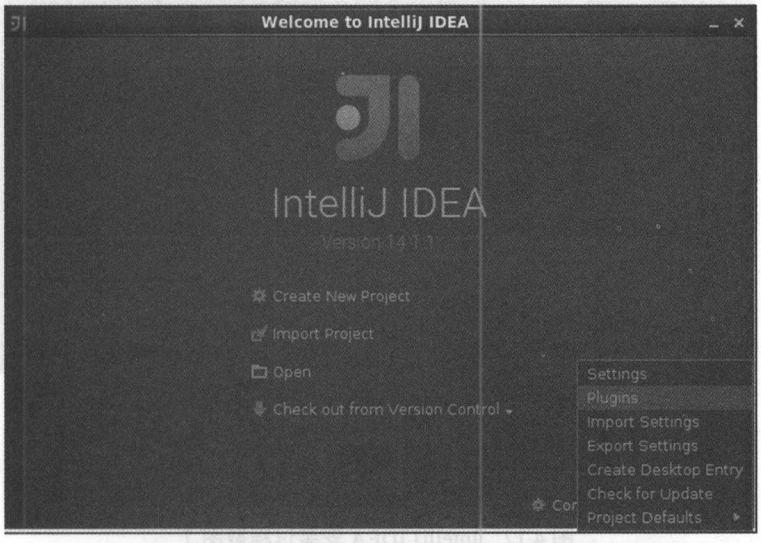


图 4.10　IntelliJ IDEA 安装过程截图 5

在弹出的窗口中选择从本地磁盘安装导入安装 Scala 插件，如图 4.11 所示。

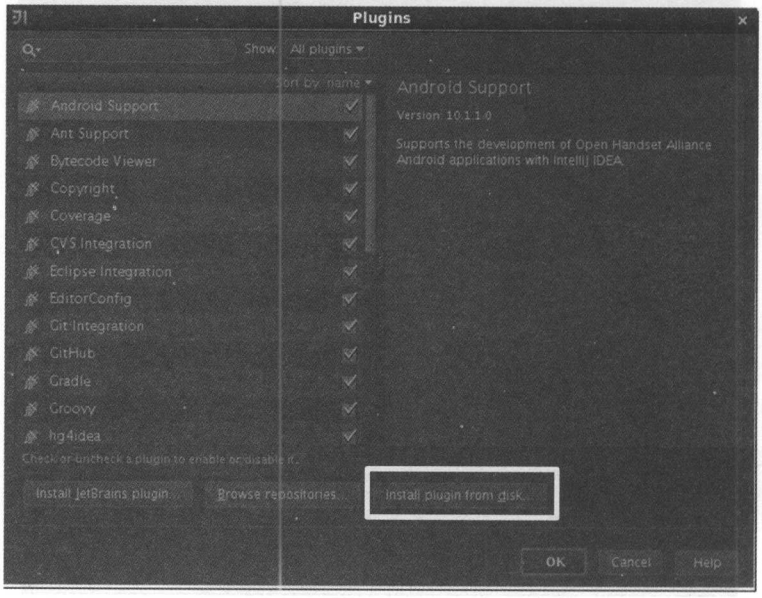


图 4.11　IntelliJ IDEA 安装过程截图 6

找到 scala-intellij-bin-1.4.15.zip 所在目录，单击导入，如图 4.12 和图 4.13 所示。

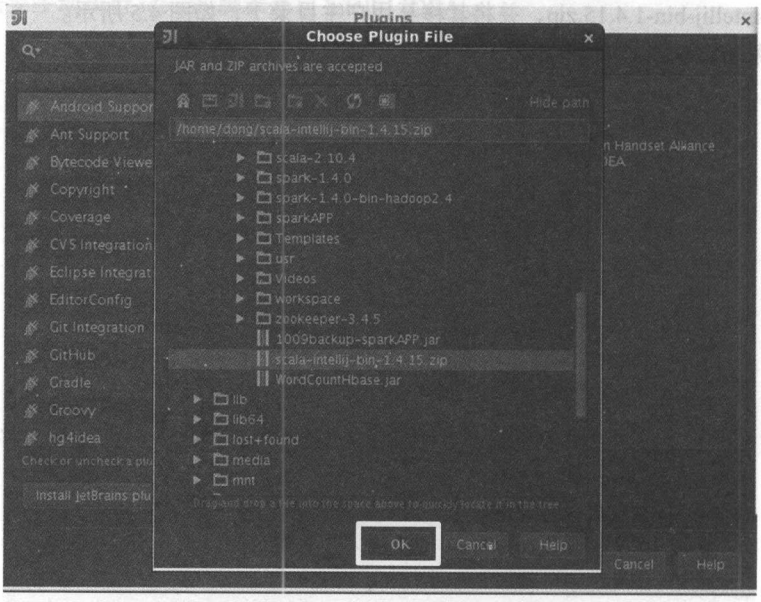


图 4.12　IntelliJ IDEA 安装过程截图 7

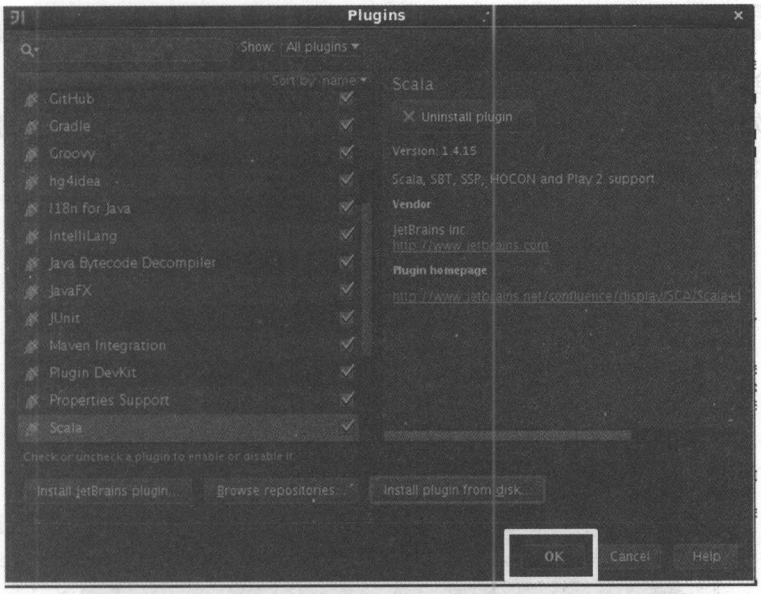


图 4.13　IntelliJ IDEA 安装过程截图 8

重新启动 IntelliJ IDEA，如图 4.14 所示。

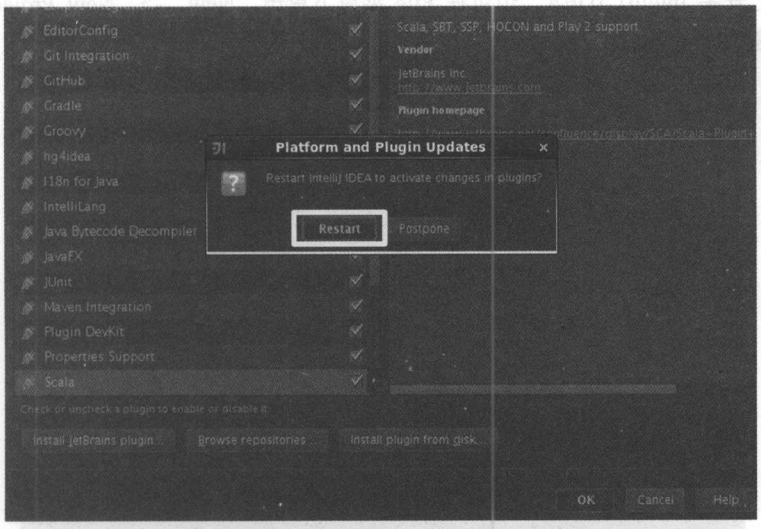


图 4.14　IntelliJ IDEA 安装过程截图 9

至此，Linux 上 IntelliJ IDEA 的搭建安装完成。

4.3.2 创建 Spark 应用程序

第 1 步，创建 Project。

如果是初次安装 IntelliJ IDEA，在弹出界面中点击“Create a New Project”直接进入 Project 创建过程，如图 4.15 所示。

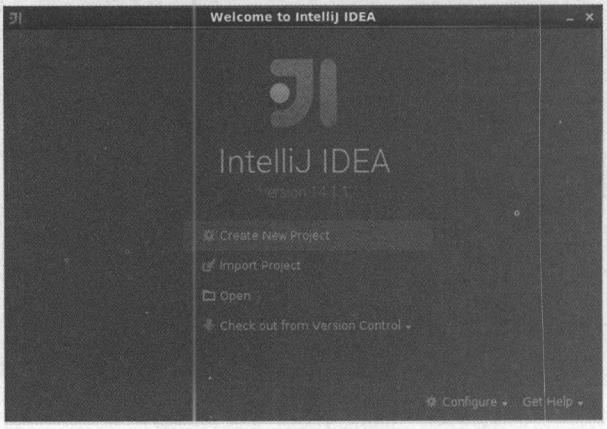


图 4.15　IntelliJ IDEA 创建 Project 过程截图 1

如果先前已安装 IntelliJ IDEA，也可在 File 菜单下单击“New” -> “New Project”命令，进入 Project 创建过程，如图 4.16 所示。

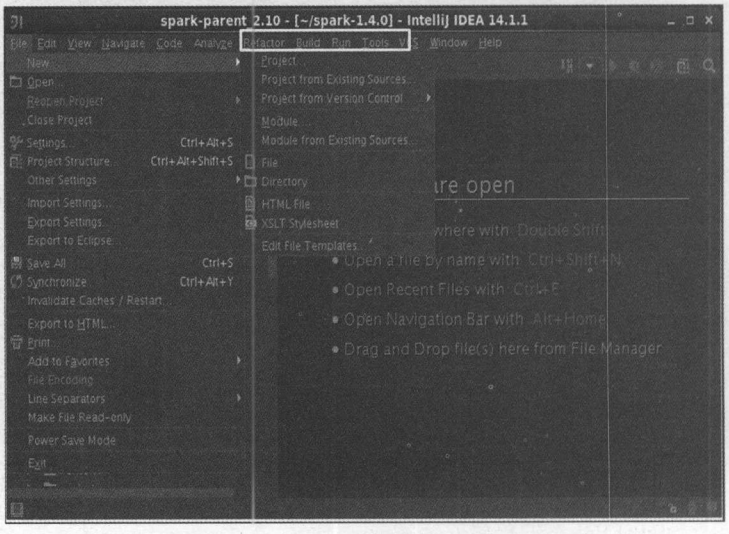


图 4.16　IntelliJ IDEA 创建 Project 过程截图 2

第 2 步，选择“Scala”，在图 4.17 所示窗口中选择“Scala”。

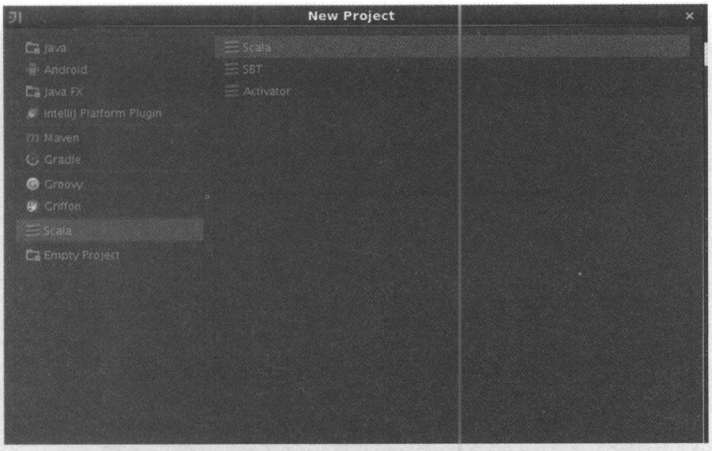


图 4.17　IntelliJ IDEA 创建 Project 过程截图 3

第 3 步，设置 Project 名称、相关目录与版本信息。

在本示例中，“Project name”设置为“SimpleApp”，“Project location”设置为“~/SimpleApp”，即与工程 SimpleApp 相关的文件都存放至该目录下，如图 4.18 所示，其中 Project SDK 需要通过“New…”设置为 JDK 的路径；Scala SDK 需要通过“Create…”设置为 Scala 的路径，如图 4.19 至图 4.22 所示。

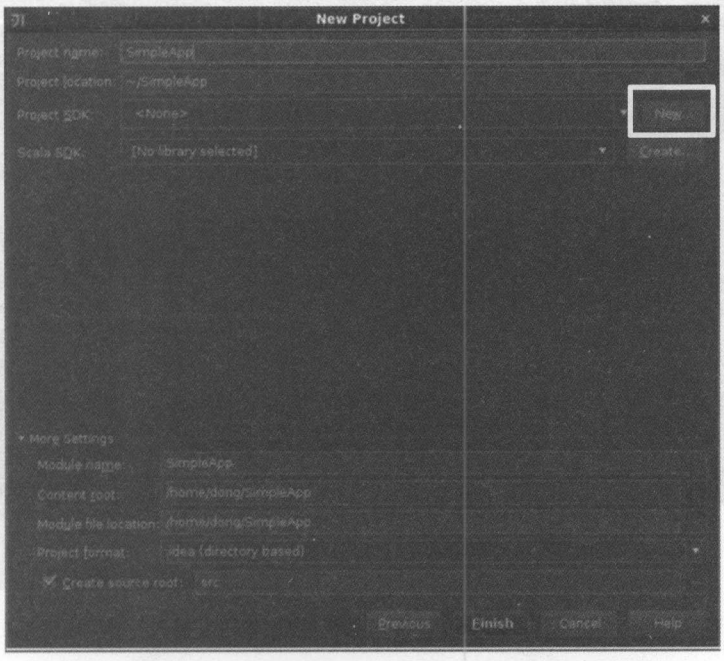


图 4.18　IntelliJ IDEA 创建 Project 过程截图 4

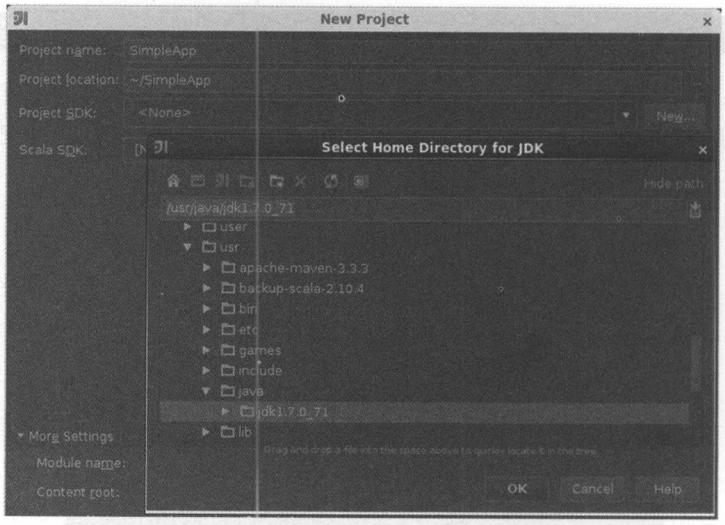


图 4.19　IntelliJ IDEA 创建 Project 过程截图 5

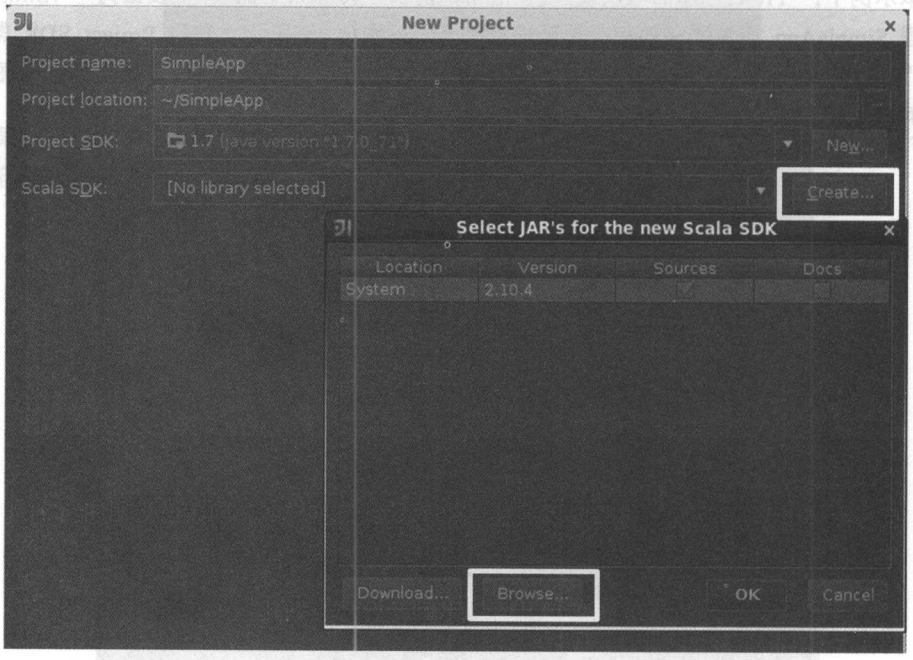


图 4.20　IntelliJ IDEA 创建 Project 过程截图 6

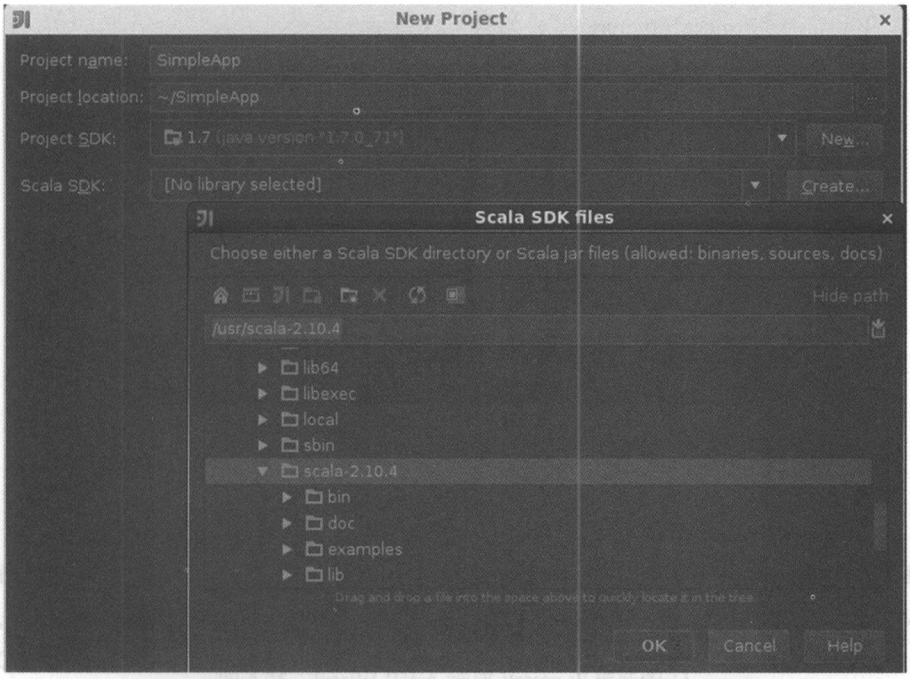


图 4.21　IntelliJ IDEA 创建 Project 过程截图 7

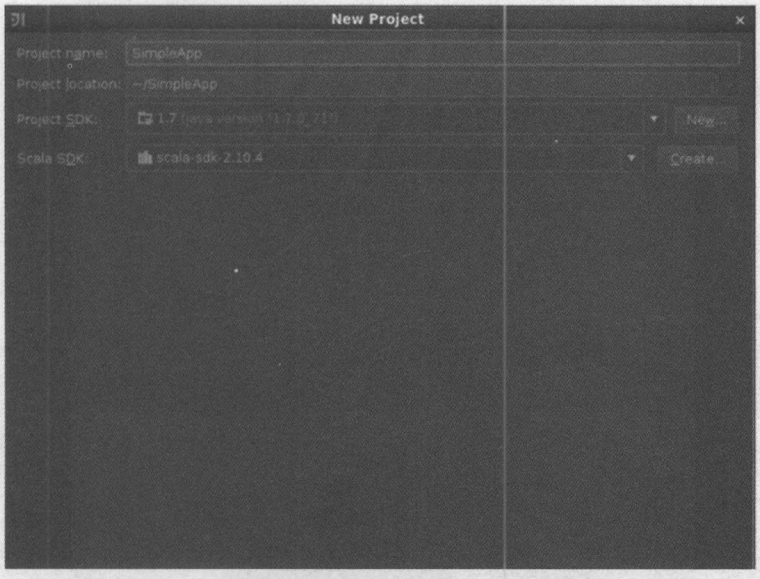


图 4.22　IntelliJ IDEA 创建 Project 过程截图 8

第 4 步，创建 Project 的目录结构。

单击“File”->“Project Structure...”，如图 4.23 所示。

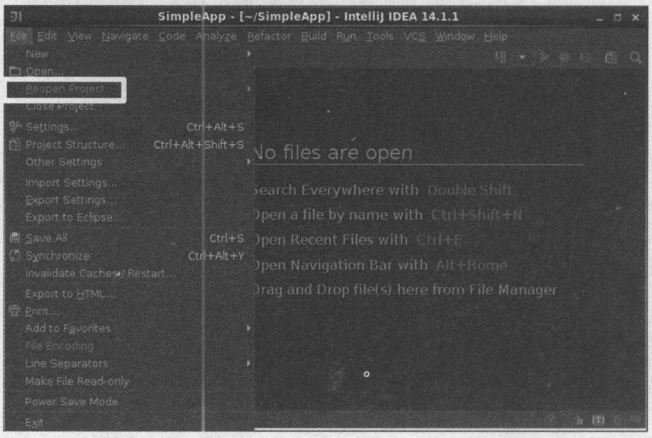


图 4.23　IntelliJ IDEA 创建 Project 过程截图 9

在弹出的窗口中单击“Modules”命令，在对应的右侧窗口中，选中“src”文件夹并单击右键，选择“New Folder...”命令，创建“main”文件夹，如图 4.24 和图 4.25 所示。同样的步骤，再选中“main”，在其中分别创建“java”、“resources”、“scala”文件夹，然后通过“Mark as”设置文件夹属性，目录结构如图 4.26 所示。

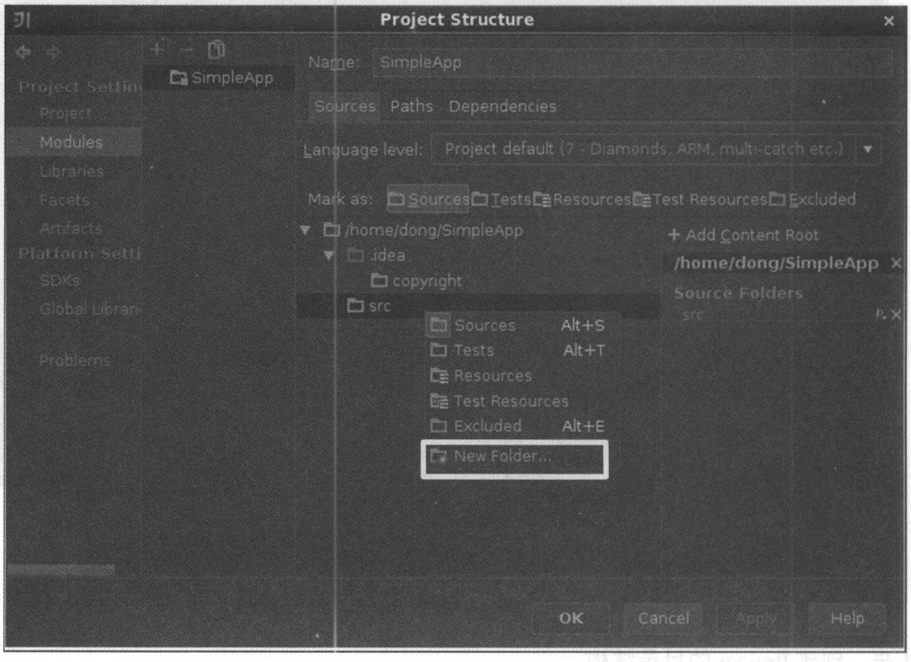


图 4.24　IntelliJ IDEA 创建 Project 过程截图 10

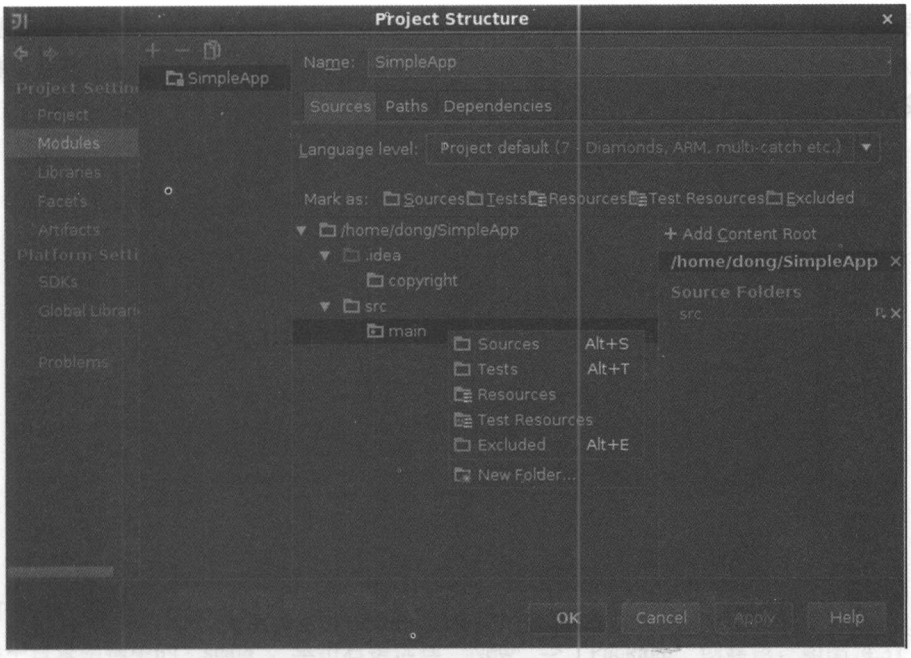


图 4.25　IntelliJ IDEA 创建 Projec 过程截图 11

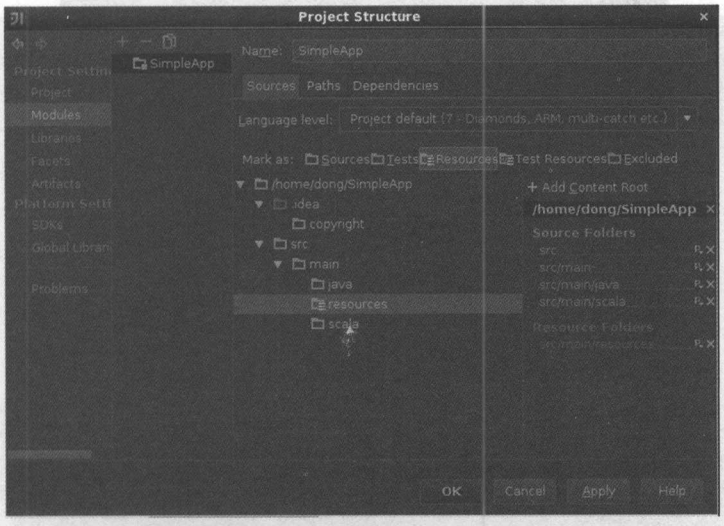


图 4.26　IntelliJ IDEA 创建 Project 过程截图 12

第 5 步，添加 Spark 依赖包。

虽然已经在 IntelliJ IDEA 中创建了 Project，但是 Project 还不能识别 Spark 类，因此需要将 Spark 相应的依赖包添加进其中。首先单击“File”->“Project Structure…”，在打开的窗口中点击“Libraries”

命令，再单击“+”，在菜单中选择“java”，如图 4.27 所示。找到安装 Spark 集群时解压出来的“spark-1.4.0-bin-hadoop2.4”文件夹，并将其 lib 目录下相应的 jar 包导入，如图 4.28 所示。依次单击“OK”按钮，分别弹出图 4.29、图 4.30 所示窗口。

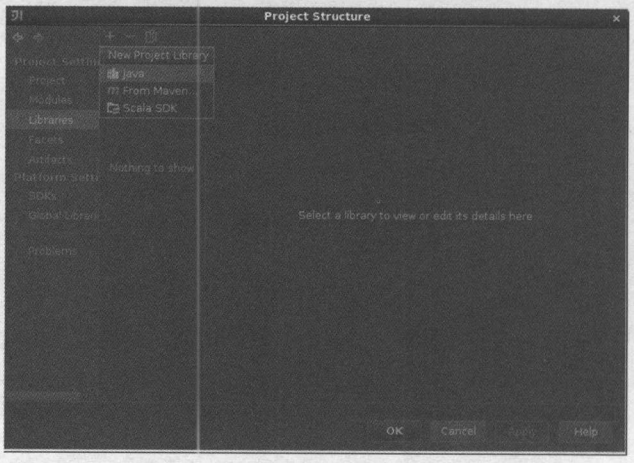


图 4.27　IntelliJ IDEA 创建 Project 过程截图 13

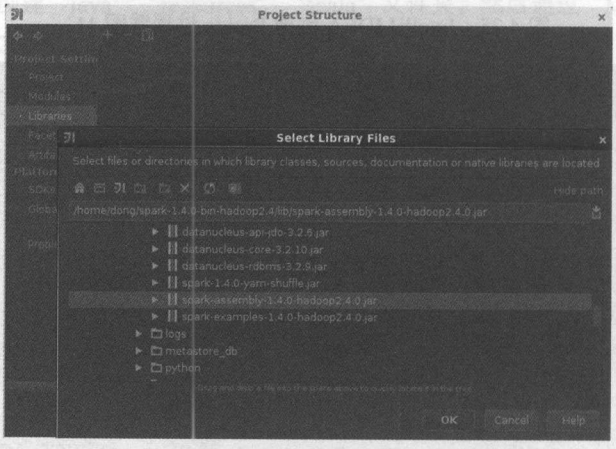


图 4.28　IntelliJ IDEA 创建 Project 过程截图 14

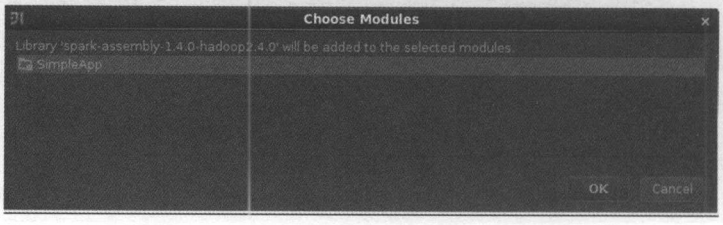


图 4.29　IntelliJ IDEA 创建 Project 过程截图 15

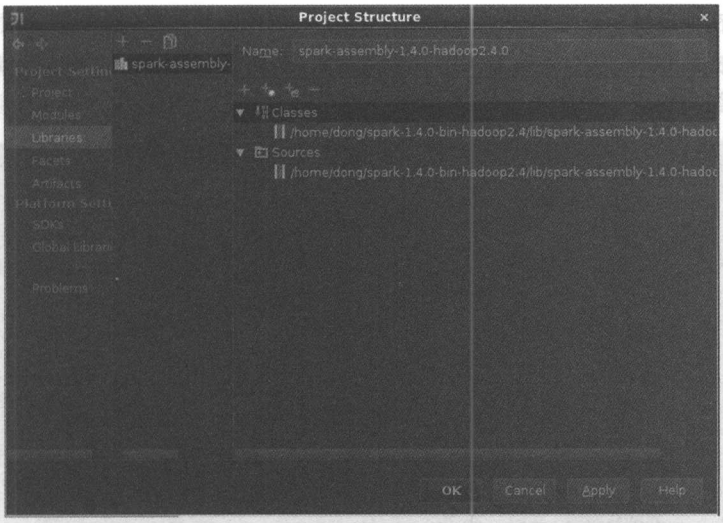


图 4.30　IntelliJ IDEA 创建 Project 过程截图 16

第 6 步，创建 Package。

选中目录结构中的“scala”，单击右键选择“New”->“Package”创建包，如图 4.31 所示。本示例中创建的包名为“dong.spark”。

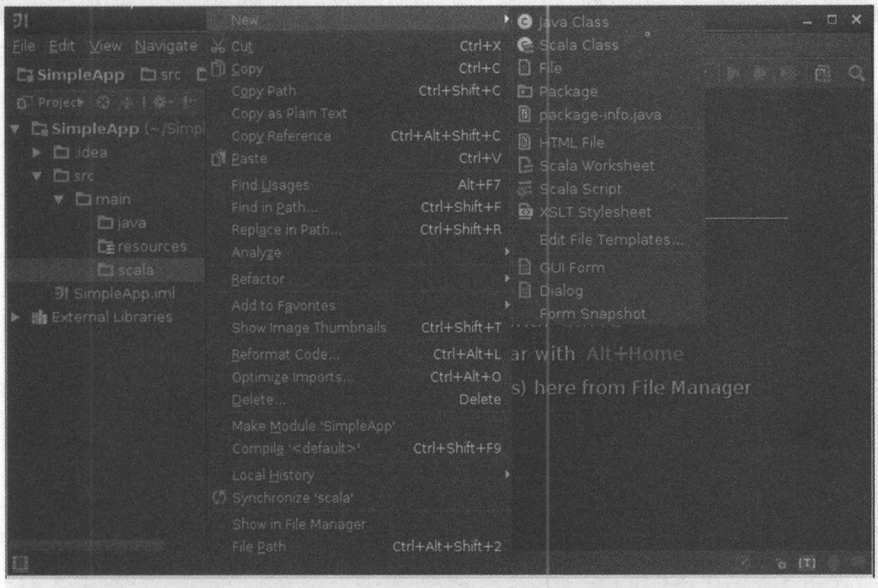


图 4.31　IntelliJ IDEA 创建 Project 过程截图 17

第 7 步，创建 Scala Object。

在左侧的 Project 目录结构中选中创建的包“dong.spark”，单击右键选择“New”->“Scala Class”，

如图 4.32 所示。在弹出的对话框中“Name”表示名称，本例设为“SimpleApp”；“Kind”表示类型，本例设为选择下拉菜单中的“Object”，如图 4.33 所示。需要注意，Scala 的语法规定 main 函数只能位于 Object 语法结构中。

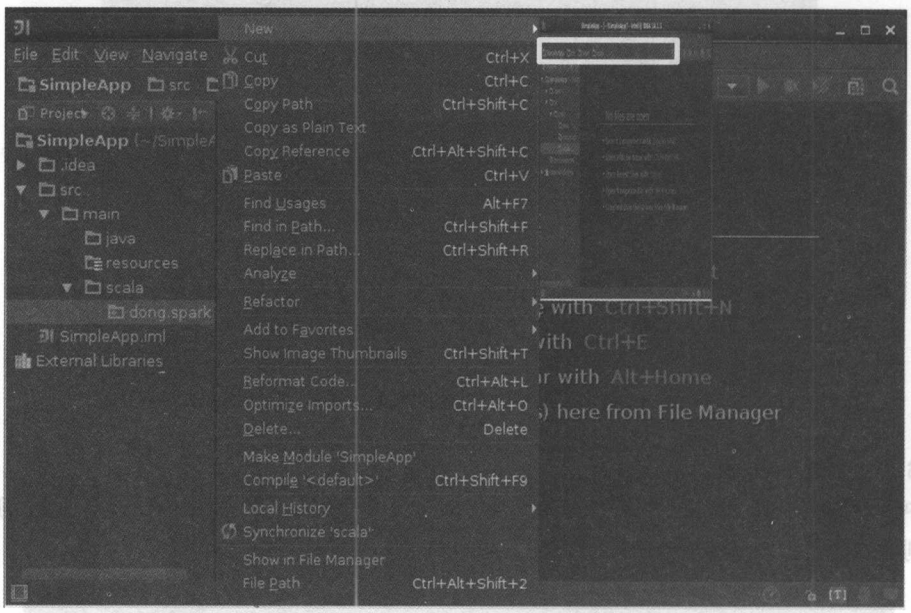


图 4.32　IntelliJ IDEA 创建 Project 过程截图 18

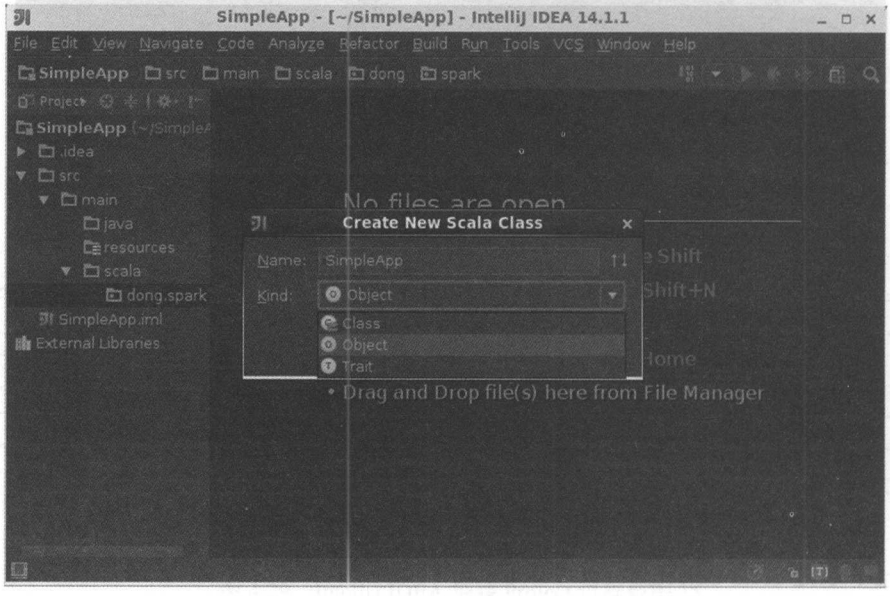


图 4.33　IntelliJ IDEA 创建 Project 过程截图 19

第 8 步，编辑 Spark 应用程序。

为方便演示和说明，将下面代码作为 SimpleApp 源文件内容。object SimpleApp 的功能是首先读取 HDFS 上指定目录下的“file.txt”文件创建 RDD；然后对 RDD 中的每个元素，即“file.txt”中的每一行利用 flatMap 操作进行切分，进而切分出若干个单词；再利用 filter 操作筛选出含有“a”的单词，并统计最终筛选出的单词的个数。

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkConf
object SimpleApp {
  def main(args: Array[String]) {
    val logFile = "hdfs://master:9000/user/dong/input/file.txt"
    val conf = new SparkConf().setAppName("Simple Application").
        setMaster("local[2]")
    //setMaster("spark://192.168.149.132:7077")
    val sc = new SparkContext(conf)
    val logData = sc.textFile(logFile, 2).cache()
    val num = logData.flatMap(x=>x.split(" ")).filter
          (_.contains("a")). count()
    println("Words with a: %s".format(num))
    sc.stop
  }
}
```

编辑后，如图 4.34 所示。

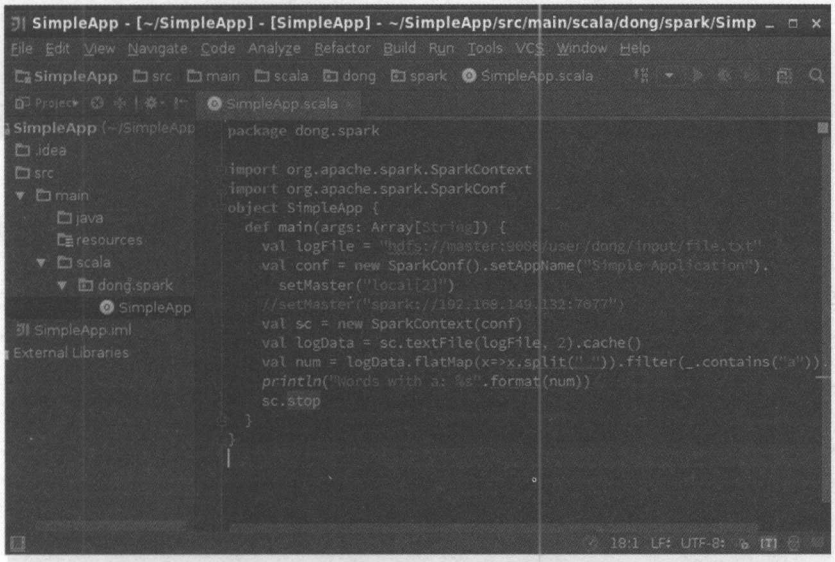


图 4.34　编辑 SimpleApp 示意图

第 9 步 本地运行 Spark 应用程序。

在图 4.34 中 setMaster()用于设置应用程序的运行方式。在该示例中 setMaster()设置为“local[2]”，表示应用程序将以本地模式运行，并使用 2 个核心。

在窗口内的空白处单击右键，选择“Run ‘SimpleApp’”，如图 4.35 所示，运行结果如图 4.36 所示。

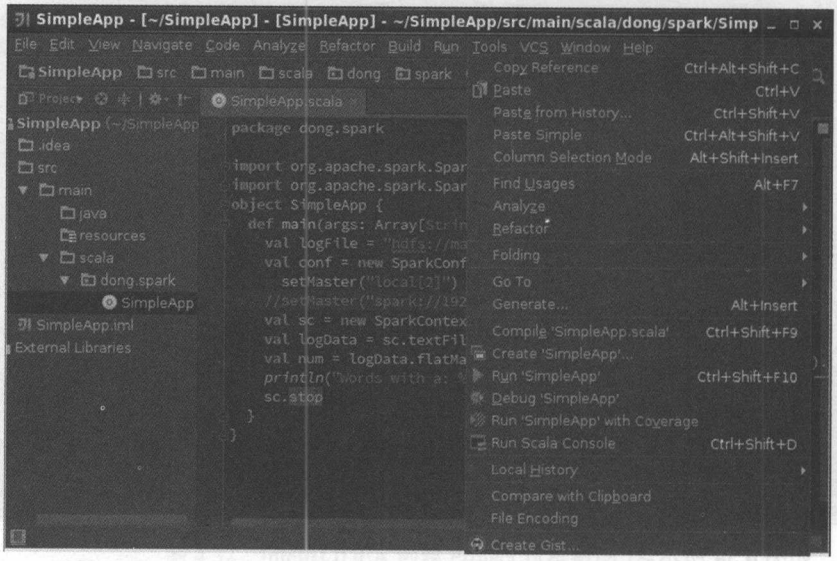


图 4.35 本地模式运行 SimpleApp 示意图

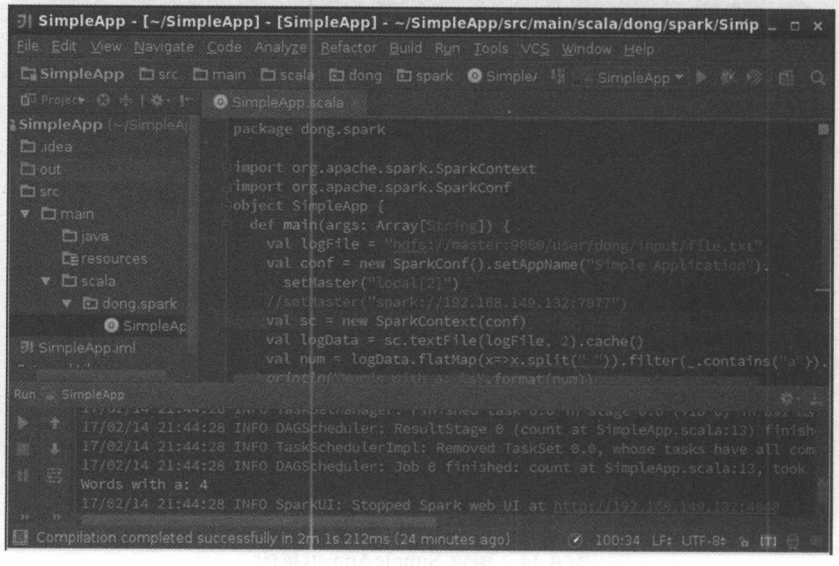


图 4.36 本地模式运行 SimpleApp 结果图

4.3.3　集群模式运行 Spark 应用程序

在 4.3.2 节中的 Spark 应用程序 SimpleApp 采用的是本地运行模式，该模式通常用于对应用程序的前期调试。以本地运行模式运行正确后，可以进一步尝试以集群模式运行应用程序。

第 1 步，设置 Spark 应用程序的启动运行类。

首先单击“File”->“Project Structure”，在打开的窗口中单击“Artifacts”命令，在右侧显示窗口中单击“+”，并在下拉菜单中选择“JAR”->“From modules from dependences...”，如图 4.37 所示。

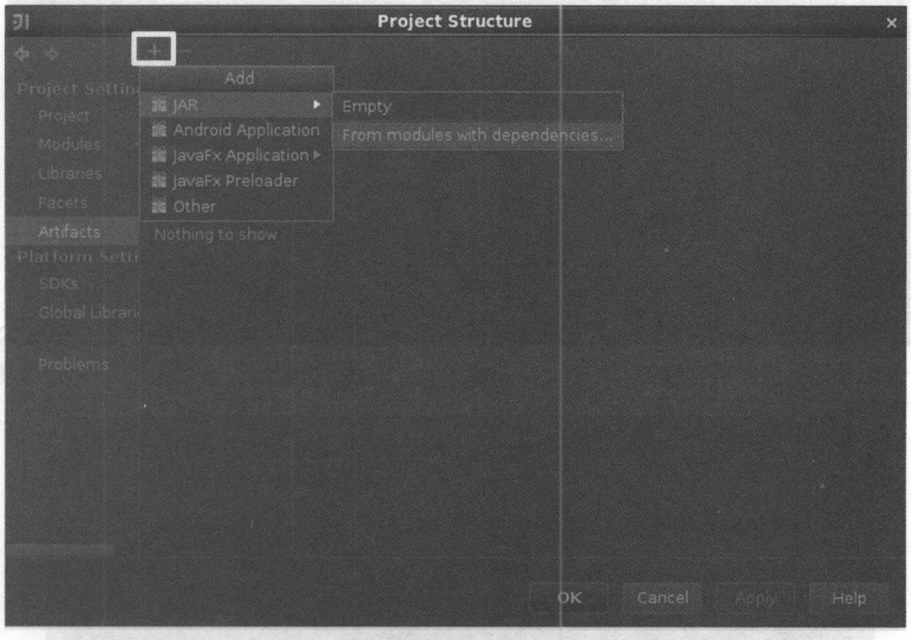


图 4.37　打包应用程序 SimpleApp 过程图 1

在弹出的对话框中设置“Main Class”，选择 SimpleApp 的启动运行类，依次如图 4.38~图 4.40 所示。

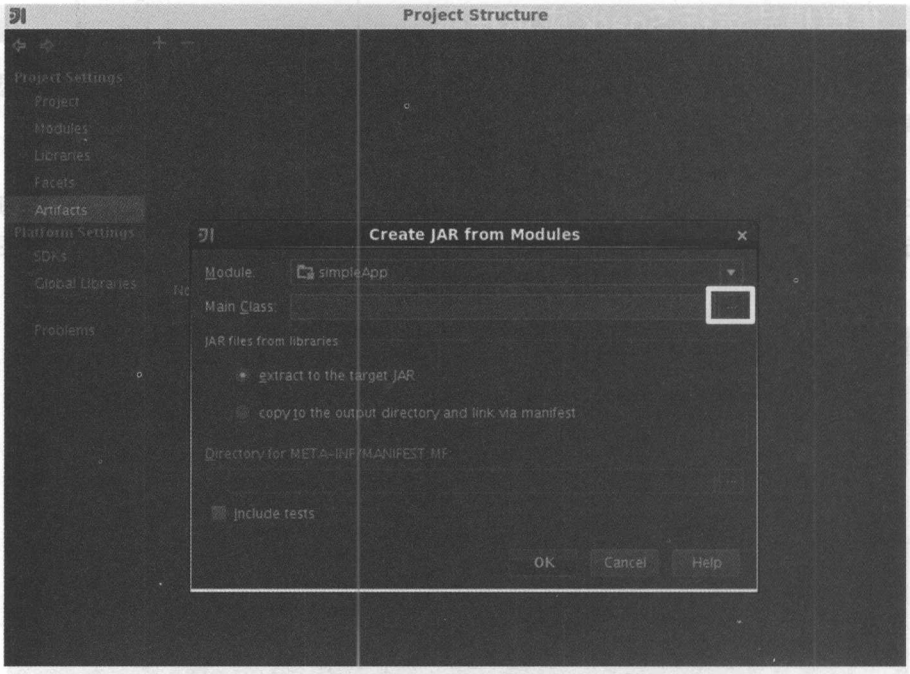


图 4.38　打包应用程序 SimpleApp 过程图 2

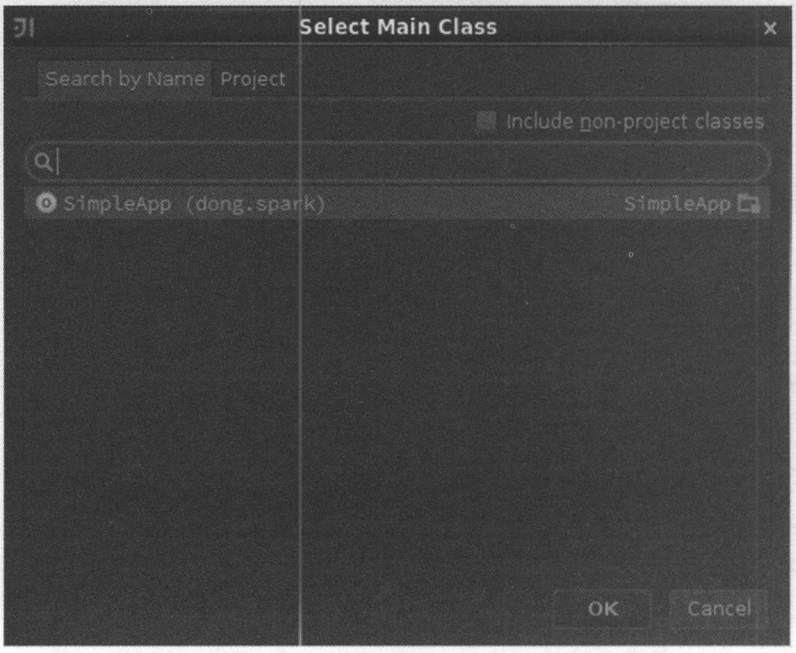


图 4.39　打包应用程序 SimpleApp 过程图 3

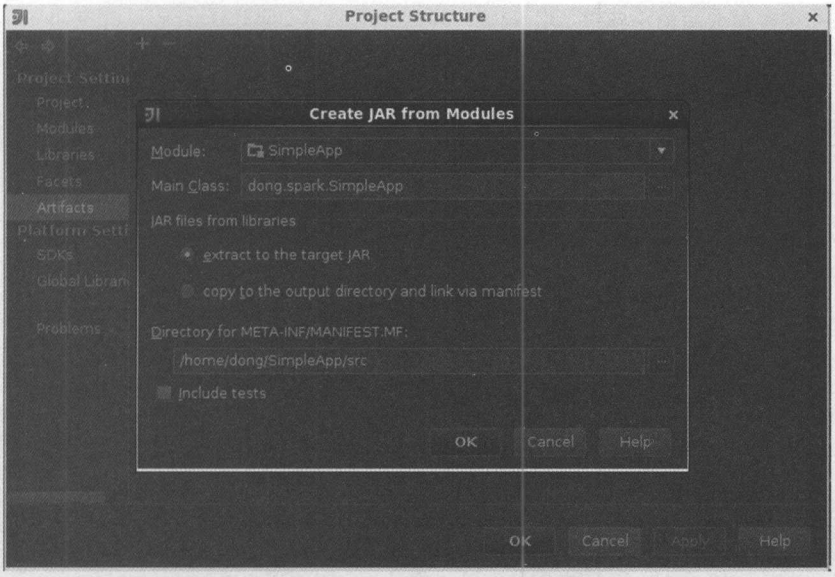


图 4.40　打包应用程序 SimpleApp 过程图 4

第 2 步，去除相应的 jar 包。

在图 4.41 所示的包中，选中除“‘SimpleApp’ compile out”之外所有的包，然后单击“-”，将其删除。

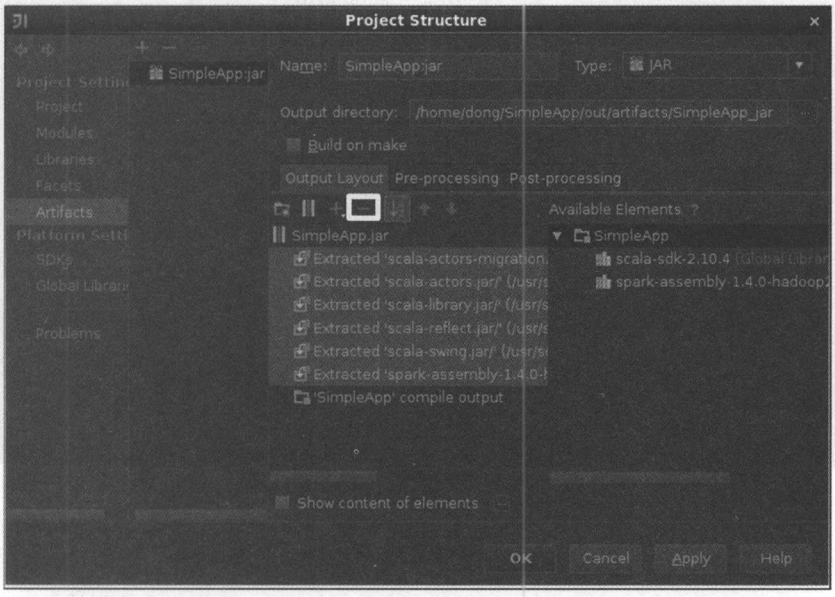


图 4.41　打包应用程序 SimpleApp 过程图 5

第 3 步，设置包名与输出路径。

为方便后续操作，更改应用程序打包后的输出路径为“/home/dong”，如图 4.42 所示。

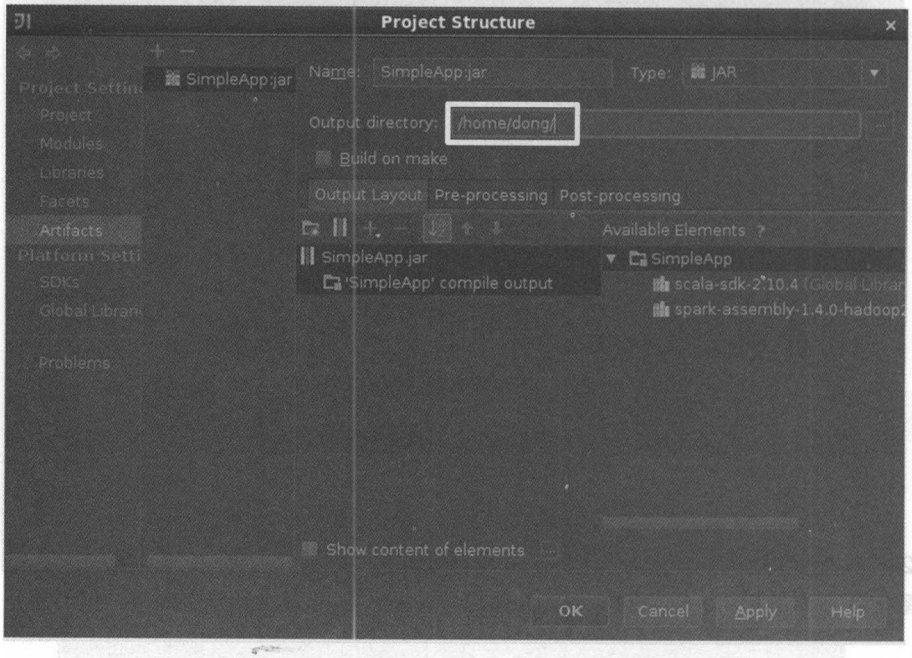


图 4.42　打包应用程序 SimpleApp 过程图 6

第 4 步，在源文件中设置运行模式。

将 SimpleApp 源文件中的 setMaster(“local[2]”)修改为 setMaster("spark://192.168.149.132:7077")，表示该应用程序将发送给 Spark 集群中该地址所标识的 Master 节点。

第 5 步，生成 jar 包。

在图 4.43 所示界面中，首先单击“Build”->“Build Artificats…”，然后在弹出的菜单中选择“SimpleApp:jar”->“Build”，如图 4.44 所示。

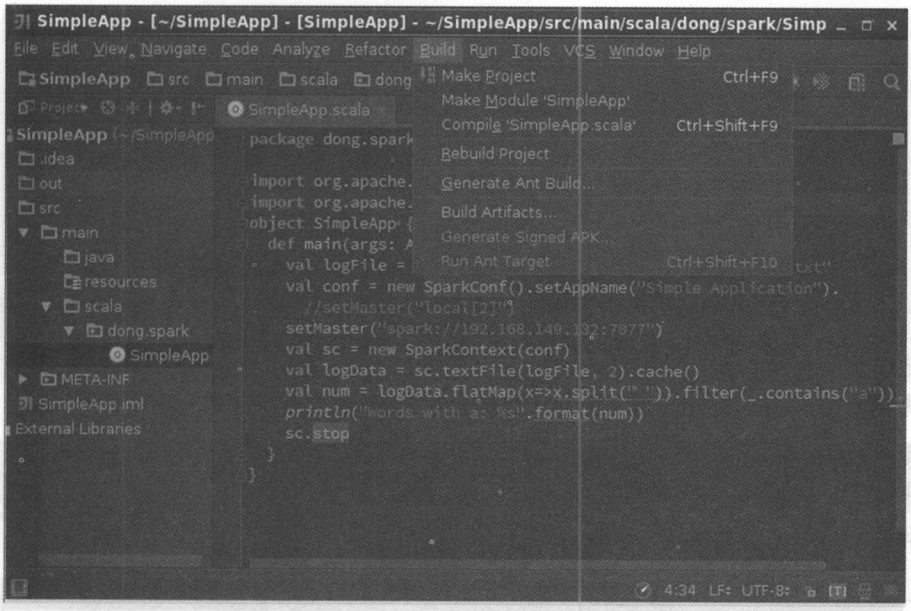


图 4.43　打包应用程序 SimpleApp 过程图 7

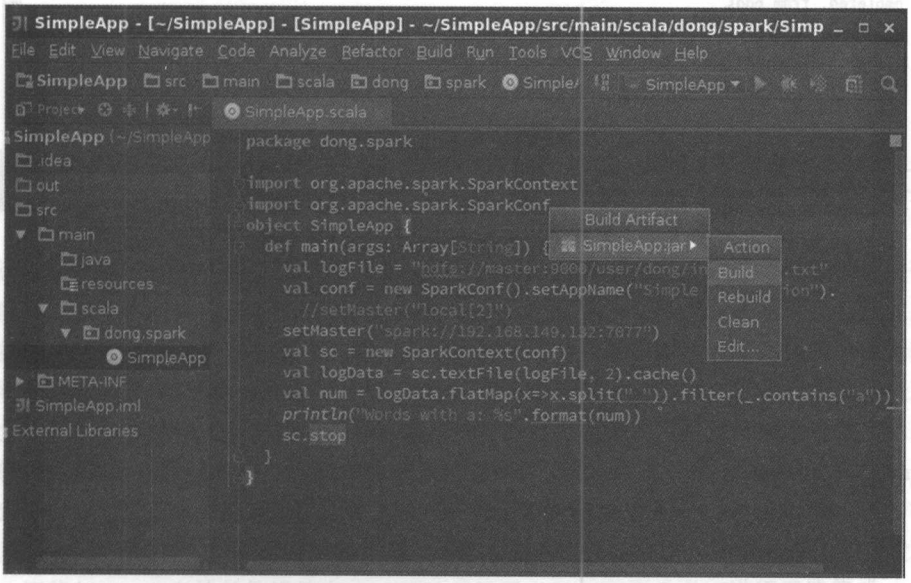


图 4.44　打包应用程序 SimpleApp 过程图 8

等待编译完成后，查看包输出路径，目录下生成的 SimpleApp.jar 文件如图 4.45 所示。

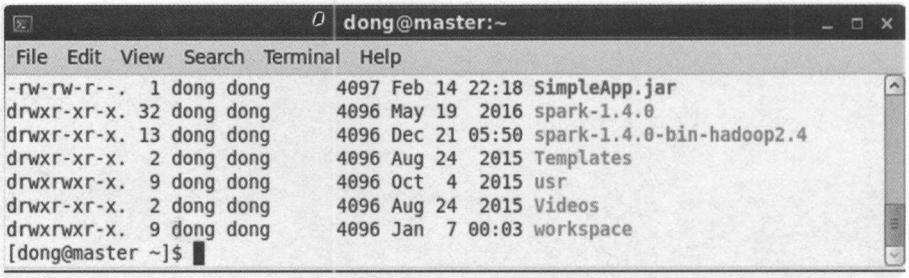


图 4.45　打包应用程序 SimpleApp 过程图 9

第 6 步，向 Spark 集群提交应用程序。

首先启动 Hadoop、Spark；然后通过下述命令向 Spark 集群提交并运行应用程序。

```
$cd spark-1.4.0-bin-hadoop2.4
$ bin/submit ~/SimpleApp.jar
```

运行结果如图 4.46 所示。

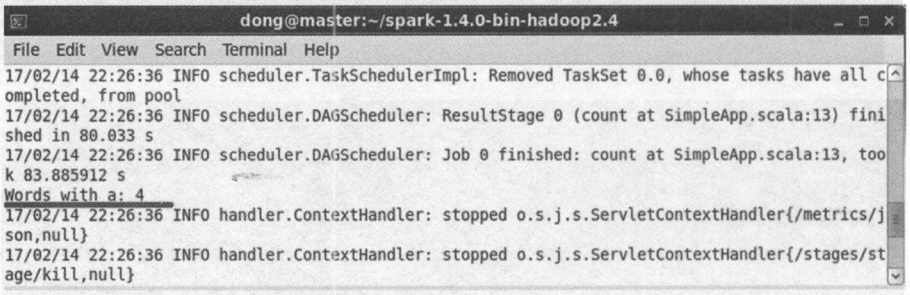


图 4.46　集群模式运行应用程序 SimpleApp 结果图

第5章

GraphX

5.1 GraphX 概述

图计算是计算机领域中的一个重要分支，在社交网络、电子商务、地图导航等领域中有着非常广泛的应用。在许多应用场景中，例如 Twitter、Facebook、微博、微信等都是大数据产生的地方，而且数据内部都存在着较高的关联度。利用图论中的理论与方法将这些关联度较高的数据转换为图数据，可以更加方便、准确地对其进行分析与处理。然而基于并行计算模式的传统大数据处理工具，对于独立性强的数据能够实现非常高的并行度与良好的可扩展性，但是对于关联度较高的图数据却面临着较大的挑战。为此涌现出一批面向大数据的图计算框架，例如 Giraph、GraphLab、GraphX 等。

本章所介绍的 GraphX 是 Spark 框架上的图计算组件。得益于 Spark 中 RDD 和 DAG 等概念，GraphX 是一种基于内存的分布式图计算模型，不仅适用于处理大数据背景下具有较多迭代次数的计算任务，还能与 Spark 上的其他组件 Spark SQL、MLlib、Spark Streaming 进行无缝集成。

但是如果仅从计算速度这一层面进行对比，GraphX 比 GraphLab 慢 2~3 倍。例如对包含 15 亿条边的图执行 PageRank 算法，Giraph、GraphX、GraphLab 性能对比如图 5.1 所示。导致 GraphX 计算性能存在差距的原因主要有两方面：GraphX 跑在 JVM 上，而 GraphLab 则是基于 C++开发的；GraphLab 不受 Spark 框架的限制，可以通过 Threads 共享内存，而 GraphX 即便在同一台机器上也有

通信开销。

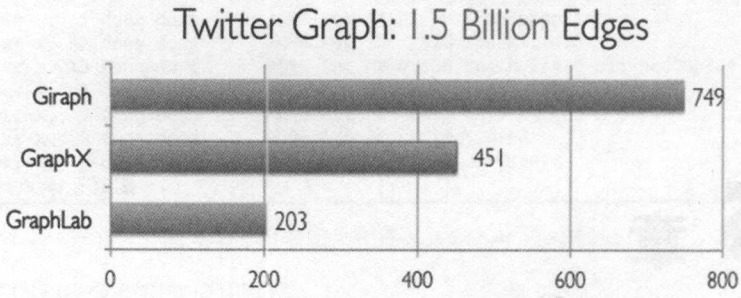


图 5.1　Graph、GraphX、GraphLab 执行 PageRank 算法性能对比图

然而，若从整个图处理与应用的流程（构建图，合并图，最终结果的查询与展示）来看，GraphX 的综合性能更具有竞争性。当面向超大规模的数据时， GraphX 运行时间的增长要慢于 GraphLab，而且其可扩展性要更好一些；从整个图计算的 Pipeline 来看，GraphX 的总体运行时间要少于 GraphLab+Spark，如图 5.2 所示。

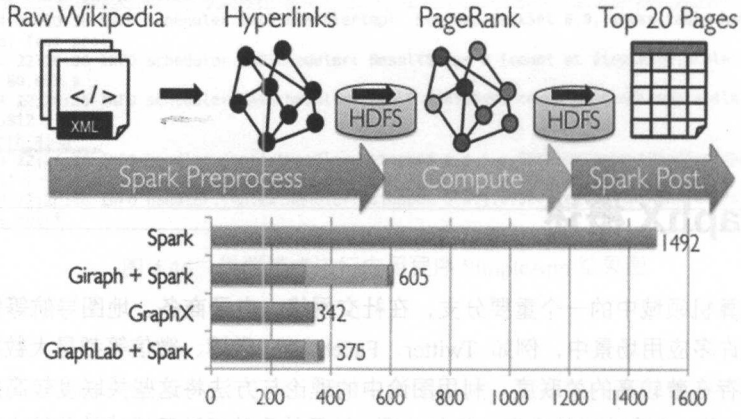


图 5.2　Spark 上 Graph、GraphX、GraphLab 的 Pipeline 性能对比

综合考虑 GraphX 的特性，越来越多的数据企业选择 GraphX 进行海量数据的图计算，例如腾讯、淘宝等。腾讯 QQ 有着国内最大的用户关系链，用户彼此间的共同好友是进行社交网络分析的一项基本指标，也是其他复杂指标的基础。腾讯借助短短百行左右的 GraphX 核心代码，在高配置的 TDW-Spark 集群上仅用两个半小时便完成了之前需要两天的全量共同好友计算。这标志着 QQ 千亿级别的关系链计算进入了小时级别时代。淘宝同样采用 GraphX 作为分布式图计算平台，实现了之前很多受限于计算能力而无法实现的模型。这些模型不仅广泛应用于用户网络中的社区发现、用户影响力评估、能量传播、标签传播等方面，提高了用户的黏性和活跃度；还应用到推荐领域中的标签推理、人群划分、年龄段预测、商品交易时序跳转等方面，提高了推荐的丰富度和准确性。

5.2 GraphX 基本原理

5.2.1 图计算模型处理流程

面向大数据的图计算模型都是对图数据进行分布式存储、并行计算和数据传输。通过操作图中顶点的数据以及与之相关联的边上的数据，产生消息并发送给相邻顶点。消息通过网络或其他方式传输，顶点在收到消息或状态后进行更新，生成新的图数据，处理流程如图 5.3 所示。

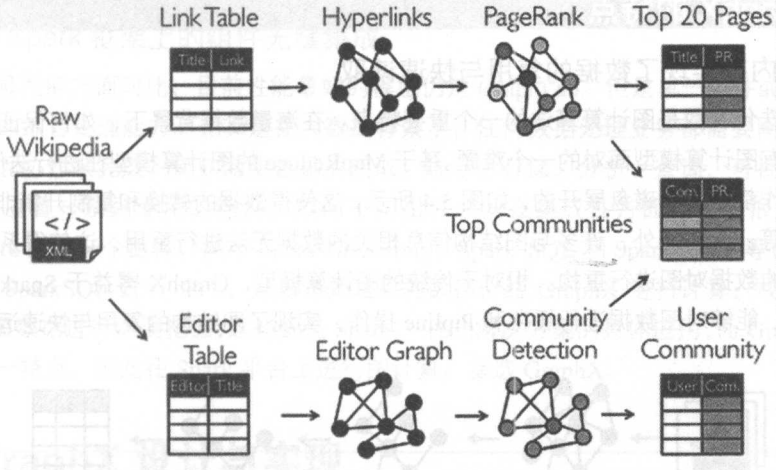


图 5.3 图计算模型处理流程

图 5.3 中上层分支描述了图计算模型的一种处理流程：首先读取 Wikipedia 原始数据；然后转换成 Link Table 形式表视图；通过分析表中蕴含的关系将其转换为 Hyperlinks 图视图；在此基础上应用 PageRank 算法对图中顶点的重要性进行评估；最后再以表的形式对计算结果进行表示与存储。图 5.3 中下层分支与上层分支类似，不同的是在转换出的图视图上应用了三角形计数用以实现社区发现，最后同样将计算结果以表的形式进行表示与存储。

由此可见，图计算模型的处理流程大致分为以下几步：

第 1 步，读取原始数据。数据既可以来自于 HDFS、Hive、Hbase、关系型数据库，也可以来自于 Spark 上的其他组件，例如 Spark Streaming、Spark SQL。

第 2 步，将原始数据转换成表视图。

第 3 步，将表视图转换成图视图。

第 4 步，在图视图上执行图计算算法。

第 5 步，将得到的结果再转换成表视图，便于表示与用户查询。

整个流程涉及了原始数据、图视图与表视图，以及两种视图的转换。GraphX 也遵循这一处理流

程，但是其与传统的大数据图计算模型相比存在着一些明显的区别。

5.2.2 GraphX 定义

GraphX 是 Spark 框架上的图计算组件，通过对 Spark 中的 RDD 进行继承与扩展，引入了弹性分布式属性图（Resilient Distributed Porperty Graph），并针对该图提供了丰富的 API。

GraphX 基于 Spark 中的 RDD、DAG、高容错性等概念和特性，实现了图计算的高效性与健壮性。GraphX 是一种基于内存的分布式的图计算库与图计算框架，用户不仅可以直接使用 GraphX 提供的经典图计算算法库，还可以针对不同的业务需求开发相应的 GraphX 应用程序。

5.2.3 GraphX 的特点

1. 基于内存实现了数据的复用与快速读取

具有较多迭代次数是图计算算法的一个重要特点。在海量数据背景下，如何保证图计算算法的执行效率是所有图计算模型面对的一个难题。基于 MapReduce 的图计算模型在进行迭代计算过程中，中间数据的操作都是基于磁盘展开的，如图 5.4 所示，这使得数据的转换和复制开销非常大，其中包括序列化开销等。除此之外，许多与图结构信息相关的数据无法进行重用，这使得系统不得不反复读取一些相同的数据对图进行重构。相对于传统的图计算模型，GraphX 得益于 Spark 中的 RDD 和任务调度策略，能够对图数据进行缓存和 Pipline 操作，实现了图结构的复用与快速运算。

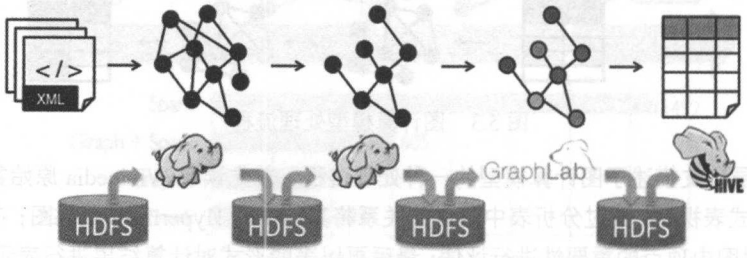


图 5.4 基于 MapReduce 的图计算模型处理流程

2. 统一了图视图与表视图

传统图计算模型都是将表视图和图视图分别进行实现的，这意味着图计算模型要针对不同的视图分别进行维护，而且视图间的转换也比较烦琐。GraphX 通过弹性分布式属性图统一了表视图和图视图，即两种视图对应同一物理存储但是各自具有独立的操作，这使得操作更具灵活性和高效性，如图 5.5 所示。基于 GraphX 执行图计算任务，一方面用户不必再对不同的组件进行学习、部署、维护和管理，降低运维成本；另一方面更有利于实现基于内存的 Pipeline 操作。

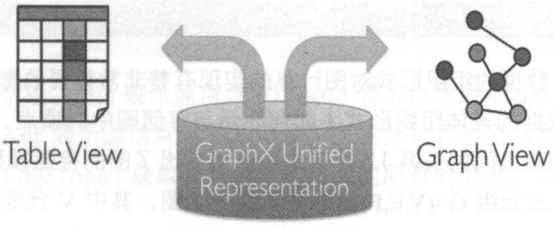


图 5.5 GraphX 中统一表视图与图视图

此外，在 GraphX 中还提出了一种路由表（routing table）。该表描述了图存储时的一些元数据信息，记录了顶点所在的物理位置以及顶点间的关系。利用该路由表能够极大地提高图计算的执行效率。

3. 能与 Spark 框架上的组件无缝集成

仅从图计算性能方面对比，目前性能最好的模型仍是 GraphLab。但是单一组件或单一性能无法决定整个系统的综合处理能力。尤其是在大数据背景下，任何数据处理业务都需要同一平台上的多个组件通过相互协作来完成，例如海量数据的获取、表示、存储、分析、查询、可视化以及数据通信等各环节对应着一系列专用的组件。然而不同组件之间在集成性方面存在着很大差异。由于 GraphX 是 Spark 上的一个组件，能与 Spark Streaming、Spark SQL 和 SparkMLlib 等进行无缝衔接，例如可以利用 SparkSQL 进行 ETL，然后将处理后的数据传给 GraphX 进行计算；或者 GraphX 与 MLlib 结合对图数据进行深度挖掘，这些都是 Spark 一栈式解决方案的具体应用，而 Giraph、GraphLab 等则不具备这一特点。因此在 Spark 平台上进行图计算，首选 GraphX。

5.3 GraphX 设计与实现

5.3.1 弹性分布式属性图

基于 Spark 中的 RDD 和图论中的相关概念，GraphX 通过弹性分布式属性图对图数据进行表示与处理，并为弹性分布式属性图提供了丰富的 API。区别于图论中的简单图，GraphX 中的弹性分布式属性图具有以下特征：

1. 图中的顶点和边都带有属性（数据）。
2. 是分布式的、基于内存的、具有较好的容错性。
3. 图中数据是高度受限的，即只可读不可写的。
4. 是有向图，即两个顶点之间的边有方向。
5. 是多重图，即两个顶点间可有多条边。

为方便说明，在不混淆的情况下，后续章节将弹性分布式属性图简称为图。

5.3.2 图的数据模型

数据是操作的对象，数据的组织形式对图计算的实现有着非常重要的影响。GraphX 定义了图的数据模型，即给出了图数据的具体组织形式（如何表示与存储图中的顶点、图中的边、顶点上的数据、边上的数据、顶点和边之间的关系），并在此基础上给出了图的存储模型和一系列并行操作。

在 GraphX 中用一个三元组 G=(V,E,P)抽象地表示一个图，其中 V 代表顶点集，E 代表边集，P 是顶点属性与边属性的集合。在这个三元组中，GraphX 用 64 位长整型作为 ID 唯一标识图中的每个顶点，因此 V 可视为一个长整型的集合；图中每一条边都关联两个顶点，因此 E 可视为顶点 ID 序对的集合；而 P 中包含的属性既可以是简单的数据类型，也可以是 *n* 元祖、列表、数组以及用户自定义的各种复杂的数据类型。

为了具体表示和存储图，GraphX 中定义了相应的类与伴生对象。

1. VertexRDD

由于图中的每一个顶点均由顶点 ID 和属性两部分构成，因此可将其形式化地描述为一个二元组(VertexId, VD)，其中 VertexId 为 64 位长整型，VD 表示顶点数据的数据类型。为了表示顶点的集合，GraphX 定义了继承自 RDD[(VertexId, VD)]的抽象类 VertexRDD[VD]和其伴生对象。

```
abstract class VertexRDD[VD] extends RDD[(VertexId, VD)]
object VertexRDD extends Serializable
```

2. Edge 与 EdgeRDD

为了表示边，GraphX 定义了样例类 Edge 和其伴生对象。

```
case class Edge[@specialized(Char, Int, Boolean, Byte, Long, Float, Double)
ED](
    var srcId: VertexId = 0
var dstId: VertexId = 0
var attr: ED = null.asInstanceOf[ED])
extends Serializable
```

Edge 中含有三个成员字段：srcId 表示源顶点 ID，是一个 64 位长整型；dstId 表示目标顶点 ID，是一个 64 位长整型；attr 表示边属性，ED 则表示边属性的数据类型。在此基础上，GraphX 将图中的每一条边用一个 Edge 类型的实例进行表示。

为了表示边集，GraphX 进一步定义了抽象类 EdgeRDD 和其伴生对象。

```
abstract class EdgeRDD[ED] extends RDD[Edge[ED]]
```

3. EdgeTriplet

EdgeTriplet 是 Edge 的子类。

```
class EdgeTriplet[VD, ED] extends Edge[ED] {
    var srcAttr: VD = _ // nullValue[VD]
    var dstAttr: VD = _ // nullValue[VD]
```

```
    ......
}
```

其包含了五个成员字段：srcId 表示源顶点 ID；dstId 表示目标顶点 ID；attr 则表示边属性；dstAttr 表示目标顶点属性；srcAttr 表示源顶点属性。VD、ED 则分别表示顶点属性和边属性的数据类型。在 GraphX 中可以通过 EdgeTriplet 获取到与这条边相关的所有的信息。顶点、边、边三元组示意图如图 5.6 所示。

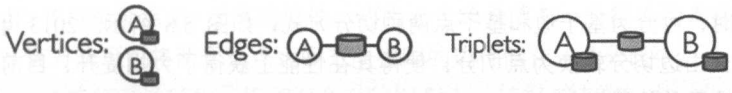
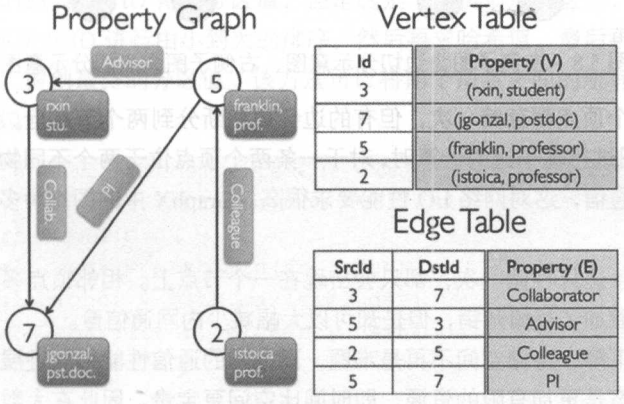


图 5.6 顶点、边、边三元组示意图

基于上述的数据模型，面向海量数据构建一张图的过程就转换为定义、创建相应 VertexRDD、EdgeRDD 等 RDD 的过程。例如图 5.7 左子图表示一张含有 4 个顶点和 4 条边的表视图，其中每个顶点代表一个人，顶点的属性表示姓名与职务，用(String, String)类型的二元组表示；每条边上的属性表示人与人之间的社会关系，用 String 类型表示；右子图是左子图对应的图视图。



Vertex Table

Id	Property (V)
3	(rxin, student)
7	(jgonzal, postdoc)
5	(franklin, professor)
2	(istoica, professor)

Edge Table

SrcId	DstId	Property (E)
3	7	Collaborator
5	3	Advisor
2	5	Colleague
5	7	PI

图 5.7 人物关系示意图

若利用 GraphX 对图 5.7 进行处理，首先需要明确其所对应的 VertexRDD、EdgeRDD 和 EdgeTriplet，分别如下：

```
    VertexRDD[(String, String)]: {(5, (franklin, professor)), (3, (rxin,
student)), (2, (istoica, professor)), …}
    EdgeRDD[String]: {Edge(5, 3, Advisor), Edge(2, 5, Colleague),..... }
    EdgeTripletRDD[(String, String), String]: {EdgeTriplet((5, (franklin,
professor)), (3, (rxin, student)), Advisor), EdgeTriplet((2, (istoica,
professor)), (5, (franklin, professor)), Colleague), … }
```

5.3.3 图的存储模型

在海量数据背景下，一张图可能包含上亿个顶点，数亿条边，甚至更多。若将 TB 级、PB 级的图数据存储在一个物理节点上并进行迭代计算通常是不现实的。GraphX 在存储方面同样采用了分布式的思想，先将一张图分解出若干子图，然后再对每张子图进行分布式存储和并行计算。因此，首先需要解决的问题就是子图的切分方式与切分策略。

1. 图的切分方式

在划分子图时，可分为基于边和基于点两种切分方式，如图 5.8 所示。2013 年，GraphLab 2.0 将子图的切分方式由边切分转换为点切分，使得其在性能上获得了大幅提升，目前基于点的切分方式已被业界广泛接受并使用。

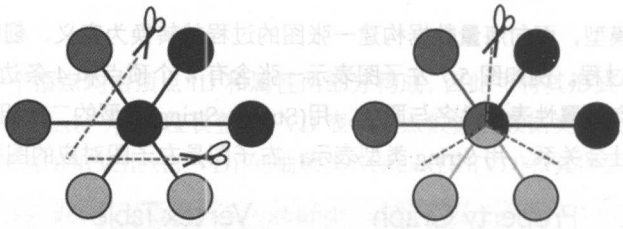


图 5.8　左侧子图为边切分示意图、右侧子图为点切分示意图

边切分：图中每个顶点都存储一次，但有的边会被切断分到两个节点上。这样做的好处是节省存储空间；缺点是对图进行基于边的计算时，对于一条两个顶点位于两个不同物理节点上的边来说，需要跨节点进行数据通信，这对网络 I/O 性能要求很高。GraphX 中的图均为多重图，使得这一缺点尤为突出。

点切分：图中每条边只存储一次，都只会出现在一个节点上。相邻顶点多的顶点可能会被复制到多个节点上，虽然增加了存储开销，但是却可以大幅减少内网通信量。

由于磁盘价格的下降，存储空间不再是难题，而内网的通信性能相对进展较慢，这使得在进行分布式计算时内网带宽是更加有限的资源，即时间比空间更宝贵。因此在大数据背景下更倾向于采用空间换时间的策略。另一方面，在绝大多数图计算应用场景中，网络都是“无尺度网络”，遵循幂律分布，不同顶点的相邻顶点数量相差非常悬殊。在这种情况下采用边切分方式会导致图计算时需要大量的数据通信，使得内网带宽更加捉襟见肘，于是边切分存储方式被渐渐抛弃了。

2. 图的切分策略

GraphX 以点切分的方式对图进行切分，并提出了 4 种具体的切分策略，其整体思路都是基于顶点 1D 确定分区值。

（1）1D 分区法

1D 分区法基于源顶点 ID 计算分区值，该方法将所有源顶点相同的边放在一个分区中。

```
case object EdgePartitionID extends PartitionStrategy {
```

```
        override def getPartition(src: VertexId, dst: VertexId, numParts:
PartitionID):
        PartitionID = {
        val mixingPrime: VertexId = 1125899906842597L
        (math.abs (src)*mixingPrime).toInt % numParts
        }
    }
```

（2）随机顶点切分法（Randowm Vertex Cut）

使用源顶点 ID 和目标顶点 ID 计算哈希值，再由这个哈希值与一个很大的素数进行模运算，得到最终的分区值。该方法可以将两个顶点之间方向相同的边存储在一个分区中。

```
    case object RandomVertexCut extends PartitionStrategy {
      override def getPartition(src: VertexId, dst: VertexId, numParts:
PartitionID): PartitionID = {
      math. abs((src, dst). hashCode ()) % numParts
      }
    }
```

（3）正则随机顶点切分法（Canonical Random Vertex Cut）

使用源顶点 ID 和目标顶点 ID 得到分区值，但是区别于随机顶点分区法，在计算哈希值的时候先将源顶点 ID 和目标顶点 ID 进行由小到大的排序，然后再求哈希值，最后再利用哈希值与一个很大的素数进行模运算，得到最终的分区值。该方法可以将两个顶点之间的所有边都放在同一个分区中，而无论边的方向如何。

```
    case object CanonicalRandomVertexCut extends PartitionStrategy {
        override def getPartition(src: VertexId, dst: VertexId, numParts:
PartitionID): PartitionID = {
            val lower = math.min(src, dst)
            val higher = math.max(src, dst)
            math.abs((lower, higher)).hashCode ()) % numParts
        }
    }
```

（4）2D 分区法（Edge Partition）

在这种方法中，通常将分区数 p 设为一个整数 t 的平方值，并且在计算分区值的时候引入了行值和列值，最终确定分区数。

```
    case object EdgePartition2D extends PartitionStrategy {
        override def getPartition(src: VertexId, dst: VertexId, numParts:
PartitionID): PartitionID = {
            val ceilSqrtNumParts: PartitionID = math.ceil(math.sqrt(numParts).
toInt
            val mixingPrime: VertexId = 1125899906842597L
```

```
            val col:
PartitionID=((math.abs(src)*mixingPrime)% ceilSqrtNumParts).tolnt
            val row:
PartitionID=((math.abs(dst)*mixingPrime))%ceiisqrtNumParts). tolnt
            (col*ceilSqrtNumParts+row)%numParts
        }
    ......
    }
```

用户可以通过 partitionBy 方法来指定一个图的切分策略。

GraphX 的分区存储示意如图 5.9 所示。

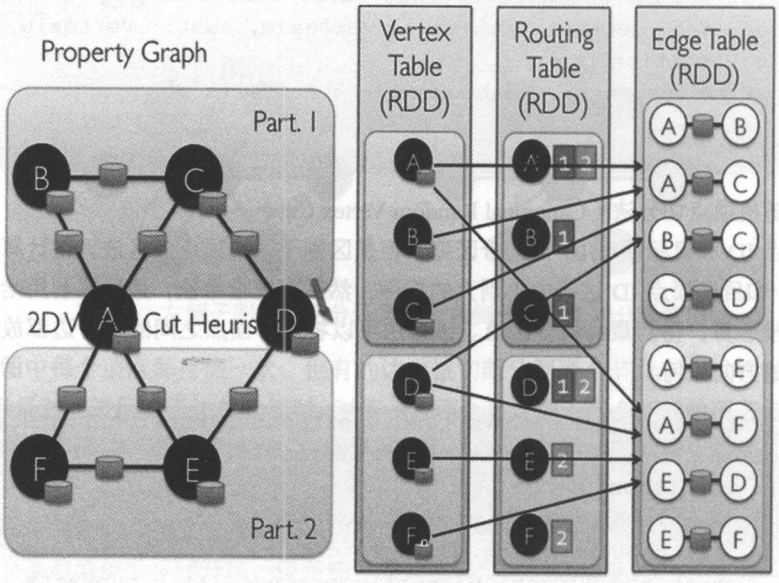


图 5.9　GraphX 分区存储示意图

图 5.9 左侧的 Property Graph 包含了 6 个顶点和 8 条边，其中 A、B、C、D 这 4 个顶点以及相应的边构成一个子图 Part.1，该子图中所有的顶点及属性对应 VertexRDD 中的蓝色分区；该子图中所有的边及属性对应 EdgeRDD 中的蓝色分区；类似的 A、D、E、F 这 4 个顶点及相应的边构成另一个子图 Part.2，该子图中所有的顶点及属性对应 VertexRDD 中的橙色分区；该子图所有的边及属性对应 EdgeRDD 中的橙色分区。与此同时，GraphX 引入了一个非常重要的概念——路由表。路由表中记录了顶点的元数据信息，例如记录了顶点 A 不仅存在于分区 1 中，还存在于分区 2 中，即 A 是两个子图的公共顶点。借助于路由表中记录的图结构信息，GraphX 在执行 mapVertices、mapEdges 等一系列操作的过程中可以对图的结构进行重用，提高执行效率。通过这个示例不难发现，GraphX 在表示和存储一张图的时候，会为这张图创建 3 个 RDD：VertexRDD、EdgeRDD、RoutingTable。如果一张图被切分成 n 个子图，那么对应这三个 RDD 会含有 n 个分区。

与 Spark 中的 RDD 一样，GraphX 中图的分布式存储使得其具有良好的容错性，当一个节点出错或者一个子图出错了，可以在集群中的其他节点上对相应的子图中进行重构。

5.3.4 GraphX 模型框架

GraphX 是非常出色的图计算框架和图计算库。用户不仅可以使用 GraphX 设计自己的图计算算法，还可以直接使用 GraphX 中提供的很多经典的图计算算法（算法库主要是通过兼容 Pregel 实现的，提供了很多 Pregel API）。GraphX 模型框架分为三个层次：算法层、模型层和实现层，如图 5.10 所示。在算法层中，主要是兼容 Pregel 的算法库；在模型层中，定义了 Graph、GraphImpl、GraphOps 三个重要的类。Graph 是抽象类，其中定义了图上的一系列基本操作；GraphImpl 是 Graph 抽象类的实现类，里面有对 Graph 中抽象方法的具体实现；GraphOps 类中不仅定义了图上的基本操作，还提供了一些经典图计算算法，例如 pageRank、triangleCount 等。GraphImpl 通过隐式转换可以调用 GraphOps 中的操作。

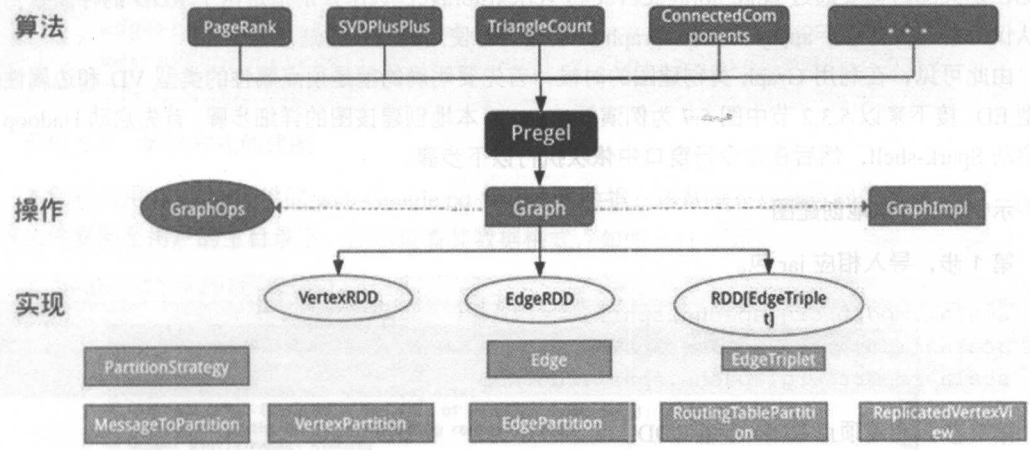


图 5.10 GraphX 模型框架图

5.4 GraphX 操作

GraphX 中对图的所有操作最终都会转换成与其相关的 RDD 的操作，也就是说对图的转换过程在逻辑上等价于对一系列 RDD 的转换过程。

5.4.1 创建图

1. 本地方式创建图

GraphX 在抽象类 Graph 的伴生对象中提供了一个实例化 Graph 类型对象的 apply 方法。

```
    def apply[VD: ClassTag, ED: ClassTag](
    vertices: RDD[(VertexId, VD)],
    edges:  RDD[Edge[ED]],
         defaultVertexAttr: VD = null.asInstanceOf[VD],
         edgeStorageLevel: StorageLevel = StorageLevel.MEMORY_ONLY,
         vertexStorageLevel:   StorageLevel   =   StorageLevel.MEMORY_ONLY):
Graph[VD, ED] = {
         GraphImpl(vertices,edges,defaultVertexAttr,edgeStorageLevel,
vertexStorageLevel)
               }
```

一个 Graph 类型对象中包含五个成员字段：字段 vertices 表示顶点集，是一个二元组 RDD，里面每个二元组为(VertexID, VD)类型，其中 VertexID 是一个 64 位长整型，而 VD 表示顶点属性的类型；字段 edges 表示边集，是一个 RDD，里面的每个元素对应一条边，每条边是一个 Edge[ED]类型的实例，其中 ED 表示边属性的类型。由 5.3.3 节可知，GraphX 中图的边与顶点对应两个不同的 RDD，因此在定义图时需要通过 edgeStorageLevel 和 vertexStorageLeverl 分别指定两个 RDD 的存储级别，默认仅存放在内存中。apply 方法是 Graph 在创建图时使用的默认方法。

由此可见，在利用 Graph 类创建图的时候，首先要明确的就是顶点属性的类型 VD 和边属性的类型 ED。接下来以 5.3.2 节中图 5.7 为例演示 GraphX 本地创建该图的详细步骤。首先启动 Hadoop，再启动 Spark-shell，然后在命令行窗口中依次执行以下步骤。

示例 5-1　本地创建图

第 1 步，导入相应 jar 包。

```
    scala>import org.apache.spark._
    scala>import org.apache.spark.graphx._
    scala>import org.apache.spark.rdd.RDD
```

第 2 步，创建顶点 RDD 与边 RDD。

图 5.7 中的顶点属性记录了姓名与职位信息，因此 VD 是（String, String）类型；边属性记录了社会关系，因此 ED 是 String 类型。在明确 VD 与 ED 后可创建相应的 RDD。

```
    scala>val      users:      RDD[(VertexId,      (String,      String))]
=sc.parallelize(Array((3L,    ("rxin",    "student")),    (7L,    ("jgonzal",
"postdoc")),(5L,("franklin", "prof")),(2L,("istoica","prof"))))
    scala>val relationships: RDD[Edge[String]]=sc.parallelize(Array(Edge(3L,
7L,"collab"), Edge(5L, 3L, "advisor"), Edge(2L, 5L, "colleague"), Edge(5L, 7L,
"pi")))
```

第 3 步，声明默认顶点。

```
    scala>val defaultUser = ("John Doe", "Missing")
```

如果存在没有属性值的顶点，则用("John Doe", "Missing")作为此类顶点的默认属性。

第4步，通过apply方法创建图。

```
scala>val graph = Graph(users, relationships, defaultUser)
```

结果：

```
graph: org.apache.spark.graphx.Graph[(String, String),String] =
org.apache.spark.graphx.impl.GraphImpl@6097d1de
```

2. 集群环境下创建图

GraphX在对象GraphLoader中提供了方法edgeListFile。该方法可以通过边集创建图，图中顶点属性与边属性的数据类型均默认为Int。

```
def edgeListFile(
    sc: SparkContext,
    path: String,
    canonicalOrientation: Boolean = false,
    numEdgePartitions: Int = -1,
    edgeStorageLevel: StorageLevel = StorageLevel.MEMORY_ONLY,
    vertexStorageLevel: StorageLevel = StorageLevel.MEMORY_ONLY)
  : Graph[Int, Int]
```

示例5-2 集群方式创建图

本例将采用Google提供的web-Google.txt作为边集数据，该数据可以通过网络下载获得。首先将该文件复制至用户的主目录下，然后查看其数据格式，如图5.11所示。

```
$ head -15 ~/web-Google.txt
```

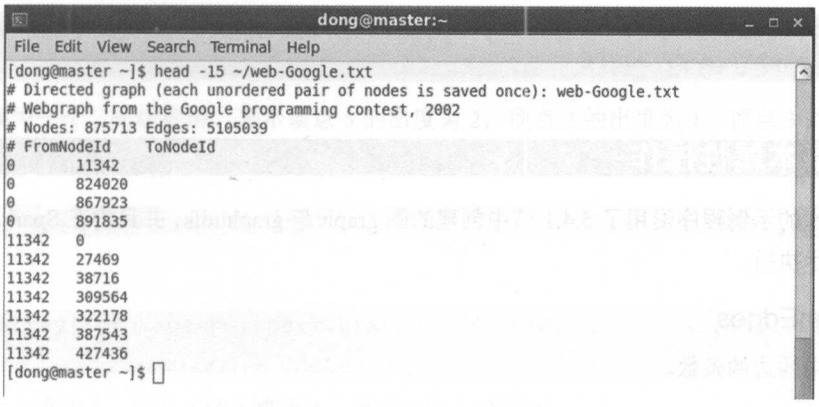


图5.11 web-Google.txt中数据格式

为方便演示，本例将仅截取出web-Google.txt中前50万行数据并另存为graphdata.txt。首先启动Hadoop与Spark；然后通过下面的代码将graphdata.txt上传到HDFS上/user/dong/input/目录中，如图5.12所示。

```
$ head -500000 ~/web-Google.txt > ~/graphdata.txt
$hdfs dfs -put ~/graphdata.txt /user/dong/input/
```

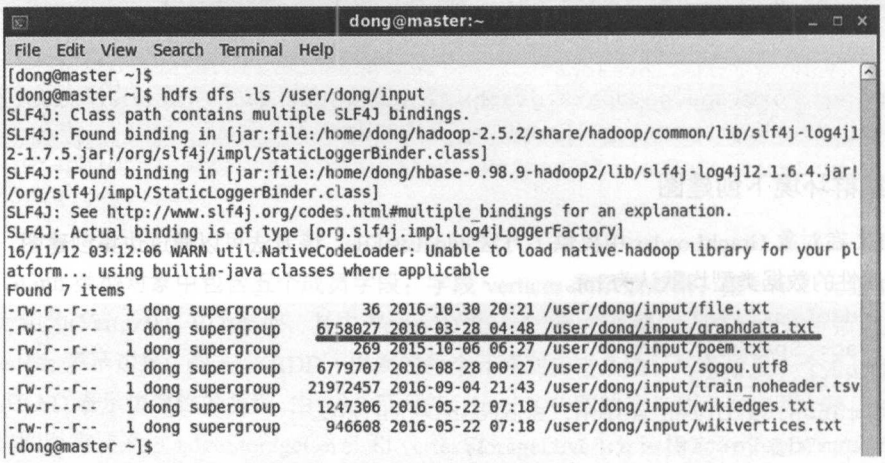


图 5.12　HDFS 上 graphdata.txt 所在目录示意图

开启 Spark-shell，在命令行窗口执行以下步骤：

第 1 步，导入相应 jar 包，与本地方式创建图中的第 1 步相同。

第 2 步，从 HDFS 读取文本文件，利用 edgeListFile 方法创建图。

```
scala>val graphhdfs=GraphLoader.edgeListFile(sc,"hdfs://master:9000/user/dong/input/graphdata.txt")
```

结果：

```
graphhdfs: org.apache.spark.graphx.Graph[Int,Int] = org.apache.spark.graphx.impl.GraphImpl@57806f6
```

5.4.2　基本属性操作

以下操作的示例程序沿用了 5.4.1 节中创建的图 graph 与 graphhdfs，并且均在 Spark-shell 的命令行窗口中演示执行。

1. numEdges

统计图中的边的条数。

示例 5-3

```
scala>graph.numEdges
```

结果：

```
res0: Long = 4
```

2. numVertices

统计图中的顶点的个数。

示例 5-4

```
scala>graph.numVertices
```

结果：

```
res1: Long = 4
```

3. degrees

统计图中顶点的度数。

示例 5-5

```
scala>graph.degrees.collect
```

结果：

```
res2: Array[(org.apache.spark.graphx.VertexId, Int)] = Array((5,3), (2,1),
(3,2), (7,2))
```

类似的，进一步查看顶点出度与入度，可分别采用方法 outDegrees、inDegrees。

示例 5-6

```
scala>graph.outDegrees.collect
```

结果：

```
res3: Array[(org.apache.spark.graphx.VertexId, Int)] = Array((5,2), (2,1),
(3,1))
```

返回结果为一个二元组数组，表示顶点 5 的出度为 2，顶点 2 的出度为 1，顶点 3 的出度为 1。

示例 5-7

```
scala>graph.inDegrees.collect
```

结果：

```
res4: Array[(org.apache.spark.graphx.VertexId, Int)] = Array((5,1), (3,1),
(7,2))
```

顶点 5 的入度为 1，顶点 3 的入度为 1，顶点 7 的入度为 2。

4. vertices

返回顶点集，即 VertexRDD[VD]。

示例 5-8

```
scala>graph.vertices.collect
```

结果：

```
res5: Array[(org.apache.spark.graphx.VertexId, (String, String))] =
Array((5,(franklin,prof)), (2,(istoica,prof)), (3,(rxin,student)),
(7,(jgonzal,postdoc)))
```

5. edges

返回边集，即 EdgeRDD[ED]。

示例 5-9

```
scala>graph.edges.collect
```

结果：

```
res6: Array[org.apache.spark.graphx.Edge[String]] = Array(Edge
(3,7,collab), Edge(5,3,advisor), Edge(2,5,colleague), Edge(5,7,pi))
```

6. triplets

返回 RDD[EdgeTriplet[VD, ED]]。

示例 5-10

```
scala>graph.triplets.collect
```

结果：

```
res7: Array[org.apache.spark.graphx.EdgeTriplet[(String, String),String]]
= Array(((3,(rxin,student)),(7,(jgonzal,postdoc)),collab),((5,(franklin,prof)),
(3,(rxin,student)),advisor),((2,(istoica,prof)),(5,(franklin,prof)),colleagu
e),((5,(franklin,prof)),(7,(jgonzal,postdoc)),pi))
```

5.4.3 结构操作

1. subgraph

返回由满足约束条件的顶点与边构成的子图。

```
def subgraph(
      epred: EdgeTriplet[VD, ED] => Boolean = (x => true),
      vpred: (VertexId, VD) => Boolean = ((v, d) => true))
    : Graph[VD, ED]
```

其中成员字段 epred 是一个匿名函数，用与以描述筛选边的约束条件；成员字段 vpred 也是一个匿名函数，用与描述筛选顶点的约束条件。

示例 5-11

```
scala>val sgraph=graph.subgraph(epred=e=>e.srcId>e.dstId)
scala>graph.edges.count
scala>sgraph.edges.count
```

将所有满足“源顶点 ID>目标顶点 ID”这一约束条件的边筛选出来构建子图 sgraph，分别统计 graph 中边的条数、sgraph 中边的条数。

结果：

```
res8: Long = 4
res9: Long = 1
```

示例 5-12

```
scala>val sgraph=graph.subgraph(vpred=(id, attr)=>id!=2)
scala>graph.vertices.count
scala>sgraph.vertices.count
```

将所有满足“顶点 ID 不等于 2”这一约束条件的顶点筛选出来构建子图 sgraph，分别统计 graph 中顶点的个数、sgraph 中顶点的个数。

结果：

```
res10: Long = 4
res11: Long = 3
```

2. reverse

将图进行反转。由于 GraphX 中的图都是有向图，例如图中顶点 5、2 之间存在一条由 2 指向 5 的边，经过 reverse 操作后，在返回的图中该边的方向变为由 5 指向 2。

示例 5-13

```
scala>val revgraph=graph.reverse
scala>revgraph.edges.collect
```

结果：

```
revgraph: org.apache.spark.graphx.Graph[(String, String),String] = org.
apache.spark.graphx.impl.GraphImpl@7195700
res12: Array[org.apache.spark.graphx.Edge[String]] = Array(Edge(7,3,
collab), Edge(3,5,advisor), Edge(5,2,colleague), Edge(7,5,pi))
```

5.4.4 转换操作

1. mapVertices

将原图中每个顶点上 VD 类型的属性进行指定的函数运算，得到新的 VD2 类型的属性，进而得

到一个新图。

```
    def mapVertices[VD2: ClassTag](map: (VertexId, VD) => VD2) (implicit eq: VD
=:= VD2 = null): Graph[VD2, ED]
```

其中 map 为一个匿名函数，描述了原图顶点属性到新图顶点属性之间的转换关系。

示例 5-14

```
    scala>graphhdfs.mapVertices((x1,x2)=>x2.toInt*3).vertices.take(10)
```

结果：

```
    res13: Array[(org.apache.spark.graphx.VertexId, Int)] = Array((536298,3),
(305874,3),  (65722,3),  (364608,3),  (475832,3),  (757908,3),  (91902,3),
(762384,3), (23776,3), (806560,3))
```

本例中 graphhdfs 经过 mapVertices 转换后得到新图，其中顶点属性为 graphhdfs 中对应顶点属性的 3 倍。转换前、后顶点的属性类型均为 Int。

2. mapEdges

将原图中每条边上 ED 类型的属性进行指定的函数运算，得到新的 ED2 类型的属性，进而得到一个新图。

```
    def mapEdges[ED2: ClassTag](map: Edge[ED] => ED2): Graph[VD, ED2]
```

其中 map 为一个匿名函数，描述了原图边属性到新图边属性之间的转换关系。

示例 5-15

```
    scala>graphhdfs.mapEdges(e=>e.attr.toInt*2).edges.take(10)
```

结果：

```
    res14: Array[org.apache.spark.graphx.Edge[Int]] = Array(Edge(0,11342,2),
Edge(0,824020,2),  Edge(0,867923,2),  Edge(0,891835,2),  Edge(1,53051,2),
Edge(1,203402,2),  Edge(1,223236,2),  Edge(1,276233,2),  Edge(1,552600,2),
Edge(1,569212,2))
```

graphhdfs 经过 mapEdges 转换后得到新图，其中边属性为 graphhdfs 中对应边属性的 2 倍。转换前、后边的属性类型均为 Int。

Graph 中还定义了 mapEdges 的重载方法：

```
    def mapEdges[ED2: ClassTag](map: (PartitionID, Iterator[Edge[ED]]) =>
Iterator[ED2]): Graph[VD, ED2]
```

两个 mapEdges 方法的功能相同，但是处理粒度不同：第一个方法每次处理一条边，而第二个方法每次处理整个分区中的边。第二个方法传递给 map 一个 partitionID 和针对该分区的 Iterator 做参数，返回包含新边值（ED2 类型）分区的 Iterator，新分区中的边与原始分区中的边是一一对应的。

3. mapTriplets

对原图中每条边的属性（ED 类型）、与该边相关联的两个顶点的属性（VD 类型）进行计算，进而生成新图。新图中顶点属性的类型仍为 VD，边属性的类型为 ED2。

```
def mapTriplets[ED2: ClassTag](map: EdgeTriplet[VD, ED] => ED2): Graph[VD, ED2]
```

其中 map 为一个匿名函数，描述了原图顶点属性、边属性到新图边属性之间的转换关系。

示例 5-16

```
scala> graphhdfs.mapTriplets(et=>et.srcAttr.toInt*2+et.dstAttr.toInt*3).triplets.take(5)
```

结果：

```
res15: Array[org.apache.spark.graphx.EdgeTriplet[Int,Int]] = Array(((0,1),(11342,1),5), ((0,1),(824020,1),5), ((0,1),(867923,1),5), ((0,1),(891835,1),5), ((1,1),(53051,1),5), ((1,1),(203402,1),5), ((1,1),(223236,1), 5),((1,1),(276233,1),5), ((1,1),(552600,1),5), ((1,1),(569212,1),5))
```

graphhdfs 经过 mapTriplets 转换后得到新图，新图中每条边的属性为 graphhdfs 中对应边所关联源顶点属性的 2 倍与目标顶点属性的 3 倍的和。转换前、后顶点与边的属性类型均为 Int。

```
def mapTriplets[ED2: ClassTag](map: (PartitionID, Iterator[EdgeTriplet[VD, ED]])=> Iterator[ED2],tripletFields: TripletFields): Graph[VD, ED2]
```

Graph 中提供了 mapTriplets 的重载方法。区别于上面的方法，重载的 mapTriplets 每次可以处理整个分区中的 EdgeTriplet 对象。

5.4.5 连接操作

1. outerJoinVertices

该操作是 GraphX 的一个核心操作，使用当前图的顶点属性和外部顶点属性进行外连接，并在连接结果上使用 mapFunc 函数进行运算，得到新图的顶点属性。

```
def outerJoinVertices[U: ClassTag, VD2: ClassTag](other: RDD[(VertexId, U)])(mapFunc: (VertexId, VD, Option[U]) => VD2)(implicit eq: VD =:= VD2 = null) : Graph[VD2, ED]
```

其中 other 是外部顶点属性的 RDD。对于原图中的每一个顶点，要求 other 中最多只包含一项与之对应的顶点属性。若 other 中不存在与原图中某个顶点对应的属性，则在生成的新图中该顶点的属性为“None”。

2. joinVertices

该操作是对 outerJoinVertices 扩展得到的，使用当前图的顶点属性和外部顶点属性进行内连接，并在连接结果上使用 mapFunc 函数进行计算，得到新图的顶点属性。

```
    def joinVertices[U: ClassTag](table: RDD[(VertexId, U)])(mapFunc: (VertexId,
VD, U) => VD): Graph[VD, ED]
```

其中 table 是外部顶点属性的 RDD。对于原图中的每一个顶点，要求 table 中最多只包含一项与之对应的顶点属性。若 table 中不存在与原图中某个顶点对应的属性，则在生成的新图中该顶点的属性沿用原图中属性。

5.4.6 聚合操作

1. aggregateMessages

对于图中的每个顶点，通过与之相邻的顶点和相关联边的属性值计算该顶点的新属性值。

```
    def aggregateMessages[A: ClassTag](sendMsg:EdgeContext[VD, ED, A] => Unit,
mergeMsg: (A, A) => A, tripletFields: TripletFields = TripletFields.All) :
VertexRDD[A]
```

其中参数 sendMsg 是一个匿名函数，描述了每个顶点向其相邻顶点传递消息的内容与方式，它是一个 EdgeContext 类型到 Unit 的一个映射（EdgeContext 中分别定义了成员方法 sendToDst 和 sendToSrc 用以向目标顶点和源顶点传递消息）；参数 mergeMsg 也是一个匿名函数，描述了每个顶点对收到消息的汇总和计算方法。

示例 5-17

```
    scala>val inDeg: RDD[(VertexId, Int)]=graph.aggregateMessages[Int](ctx =>
ctx.sendToDst(1), _ + _)
    scala>inDeg.collect
```

结果：

```
    res16: Array[(org.apache.spark.graphx.VertexId, Int)] = Array((5,1), (3,1),
(7,2))
```

本示例借助于 aggregateMessages 操作实现了统计 5.4.1 节中 graph 的每个顶点的度数。aggregateMessages 中第一个参数是一个匿名函数，其中 ctx 表示一个 EdgeContex 类型的实例，通过该实例调用其成员方法 sendToDst 向目标顶点发送消息整数 1。第二个参数也是一个匿名函数，表示每个顶点对汇总后的整数 1 进行累加操作。

5.5 GraphX 案例解析

5.5.1 PageRank 算法与案例解析

1. 算法原理

PageRank 算法即网页排名算法，是 Google 创始人拉里·佩奇和谢尔盖·布林于 1997 年构建早期的搜索系统原型时提出的链接分析算法。自从 Google 在商业上获得巨大成功后，该算法引起了研究者们的广泛关注，目前很多重要的链接分析算法都是在 PageRank 算法基础上衍生出来的。PageRank 算法是 Google 用来标识网页等级的重要依据，是 Google 衡量一个网站的好坏的唯一标准。对网页进行排名需要量化的依据，因此 PageRank 算法对每一个网页进行计算后会得到一个在 0 到 10 范围内的值，即该网页的 PageRank 值，简称 PR 值。PR 值越高说明网页越受欢迎，越重要。PageRank 算法的核心步骤如下：

第 1 步，初始化。

PageRank 算法基于两个假设：数量假设和质量假设。首先通过链接关系构建 Web 图（网络中每个页面对应 Web 图中的一个顶点，若网页 A 中包含一条指向 B 的链接，则 Web 图中存在一条由顶点 A 指向顶点 B 的边）；然后为 Web 图中的每个顶点设置初始的 PR 值（通常设定每个顶点的初始 PR 值为 1/N，其中 N 为网络中网页的个数）。

第 2 步，迭代计算。

首先假设一个用户在访问某网页时，其将跳转到该网页上各超链接页面的概率相同。例如图 5.13 中网页 A 链向网页 B、C、D，所以根据假设，用户从 A 跳转到 B、C、D 的概率各为 1/3。PageRank 算法在每一轮迭代计算的过程中，将每个网页当前的 PR 值平均分配到该网页指向的超链接页面上，这样每个网页便获得了相应的权值；再将新权值求和，即得到该网页的新 PR 值。当每个网页的 PR 值都获得更新后，就完成了一轮迭代。

第 3 步，结束。

随着每一轮的迭代计算，网页的 PR 值会不断得到更新。当迭代达到一定次数，或者每个网页的 PR 值固定不变，再或者 PR 值收敛至某一范围内时，算法停止。算法停止时每个网页的 PR 值就是该网页最终的 PR 值。

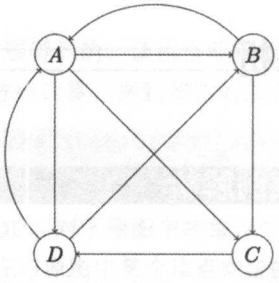


图 5.13　由四个网页及其超链接关系构成的 Web 图

GraphX 提供了静态和动态 PageRank 的实现方法，这些方法在 PageRank 对象中。静态的 PageRank 运行固定次数的迭代，而动态的 PageRank 一直运行直到收敛为止。

2. 案例分析

GraphX 提供了一个运用 PageRank 算法分析社交网络中各用户重要性的案例。社交网络中的用户数据在 spark-1.4.0/graphx/data/users.txt 中，用户之间关系数据在 spark-1.4.0/graphx/data/followers.txt 中。

users.txt 内容如下：

```
1,BarackObama,Barack Obama
2,ladygaga,Goddess of Love
3,jeresig,John Resig
4,justinbieber,Justin Bieber
6,matei_zaharia,Matei Zaharia
7,odersky,Martin Odersky
8,anonsys
```

followers.txt 内容如下：

```
2 1
4 1
1 2
6 3
7 3
7 6
6 7
3 7
```

首先以 users.txt 中的用户作为顶点、followers.txt 中的关系作为边集创建图；然后通过图直接调用 PageRank 算法计算出每个顶点的 PR 值，即用户的重要性；最后结合用户的属性信息对结果输出展示。

示例 5-18 基于 IntelliJ IDEA 的 PageRank 算法应用实例

```
package dong.spark
import org.apache.log4j.{Level, Logger}
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf
import org.apache.spark.graphx._
object PageRankTest {
  Logger.getLogger("org.apache.spark").setLevel(Level.WARN)
  Logger.getLogger("org.eclipse.jetty.server").setLevel(Level.OFF)
  def main(args: Array[String]) {
    val conf=new SparkConf().setAppName("PageRankTest").
             setMaster("local[2]")
    val sc = new SparkContext(conf)
//通过边集 followers.txt 创建图 graph
   val graph = GraphLoader.edgeListFile(sc,"/home/dong/spark-1.4.0/
graphx/data/followers.txt")
/*在 graph 上运行 PageRank 算法返回带有 PR 值的新图；继续调用 vertices 方法返回新图中
的顶点集合 ranks*/
val ranks = graph.pageRank(0.001).vertices
//创建顶点 RDD[(VertexID, String)]，用切分出的字符串作为顶点属性。
   val users = sc.textFile("/home/dong/spark-1.4.0/graphx/data/users.
txt").
                map{ line =>
                val fields = line.split(",")
                (fields(0).toLong, fields(1))
    }
//users 与 ranks 进行连接操作，返回由用户名称、用户 PR 构成的 RDD。
   val ranksByUsername = users.join(ranks).map {
case (id, (username, rank)) => (username, rank)
    }
//打印结果
   println(ranksByUsername.collect().mkString("\n"))
  }
}
```

运行结果，如图 5.14 所示。

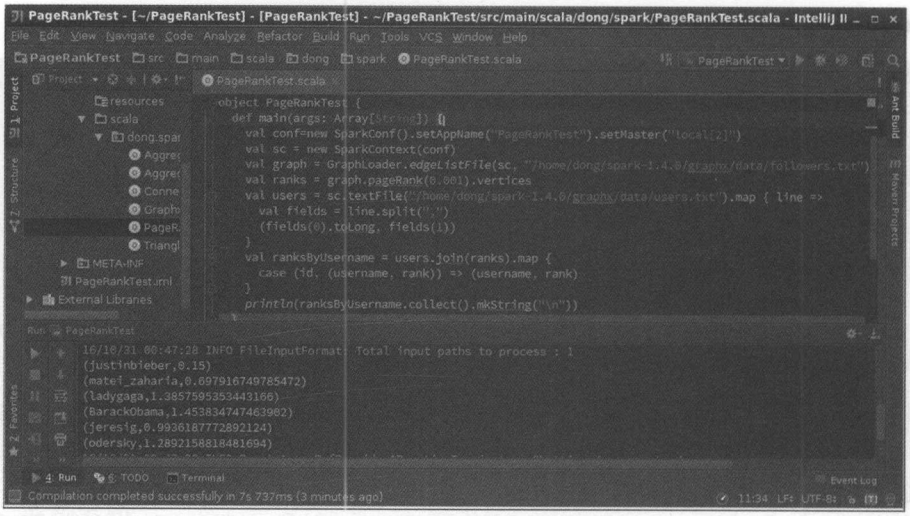


图 5.14　PageRankTest 运行结果

5.5.2　Triangle Count 算法与案例解析

Triangle Count（三角形计数）在社交网络分析中非常有用。三角形是一个由三个顶点构成的小图，其中顶点两两相连。例如在 Facebook 上一个用户认识两个好友，而这两个好友彼此相识，那么这三个用户就组成了一个三角形。一个社交网络中拥有这样的三角形越多，就意味着网络中用户联系越紧密。通过三角形计数能够计算出集群的度，这是进行聚类分析的重要依据和指标。

1．算法原理

一个顶点有两个相邻的顶点并且相邻顶点之间存在边时，这个顶点是一个三角形的一部分。如果顶点 i 到顶点 j 既存在长度为 1 的路径，又存在长度为 2 的路径（假设经过的中间顶点是 k），那么顶点 i、j、k 就可以构成一个三角形。利用有向图的邻接矩阵 A 构造矩阵 A^k，其中元素 a_{ij} 表示从顶点 i 经过 k 路径到达顶点 j 的通路总数。因此 A 和 A^2 相同位置非零值的个数即为图中三角形的个数，用公式表示为：

$$\text{TriangleCount}=\text{nnz}(A\cap A^2)$$

例如图 5.15 是一个拥有四个顶点的有向图，其邻接矩阵 A 和 A^2 分别如图 5.16 所示，因此可判断出图中共含有 2 个三角形。

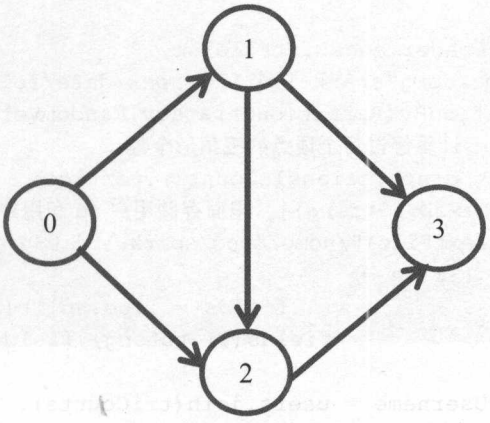


图 5.15 四个顶点的有向图

$$A=\begin{bmatrix}0&1&1&0\\0&0&1&1\\0&0&0&1\\0&0&0&0\end{bmatrix},\quad A^2=\begin{bmatrix}0&0&1&2\\0&0&0&1\\0&0&0&0\\0&0&0&0\end{bmatrix}$$

http://blog.csdn.net/

图 5.16 对应的邻接矩阵 a 与 a^2

注意：GraphX 中三角形计数算法要求边的方向是规范的方向，即源顶点 ID 小于目标顶点 ID。

2. 案例分析

GraphX 提供了一个运用三角形计数算法的案例，即统计社交网络中每个用户构成三角形的个数。社交网络中的用户数据、用户之间关系数据仍沿用 5.5.1 中的数据。

示例 5-19

```
import org.apache.spark.SparkContext
import org.apache.log4j.{Level, Logger}
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf
import org.apache.spark.graphx._
object TriangleCount {
Logger.getLogger("org.apache.spark").setLevel(Level.WARN)
 Logger.getLogger("org.eclipse.jetty.server").setLevel(Level.OFF)
def main(args: Array[String]) {
   val conf = new SparkConf().setAppName("TriangleCount").
              setMaster("local[2]")
```

```
    val sc = new SparkContext(conf)
//加载数据构建图。
    valgraph=GraphLoader.edgeListFile(sc,
            "/home/dong/spark-1.4.0/graphx/data/followers.txt",true).
            partitionBy(PartitionStrategy.RandomVertexCut)
//通过 triangleCount 计算经过每个顶点的三角形个数
    val triCounts = graph.triangleCount().vertices
//创建用户 RDD[(VertexID, String)]，里面存储用户 id 和用户名信息
    val users = sc.textFile("/home/dong/spark-1.4.0/graphx/data/users.txt").
              map { line =>
                          val fields = line.split(",")
                          (fields(0).toLong, fields(1))
                }
    val triCountByUsername = users.join(triCounts).
              map { case (id, (username, tc)) =>
                          (username, tc)
                    }
    println(triCountByUsername.collect().mkString("\n"))
  }
}
```

运行结果，如图 5-17 所示。

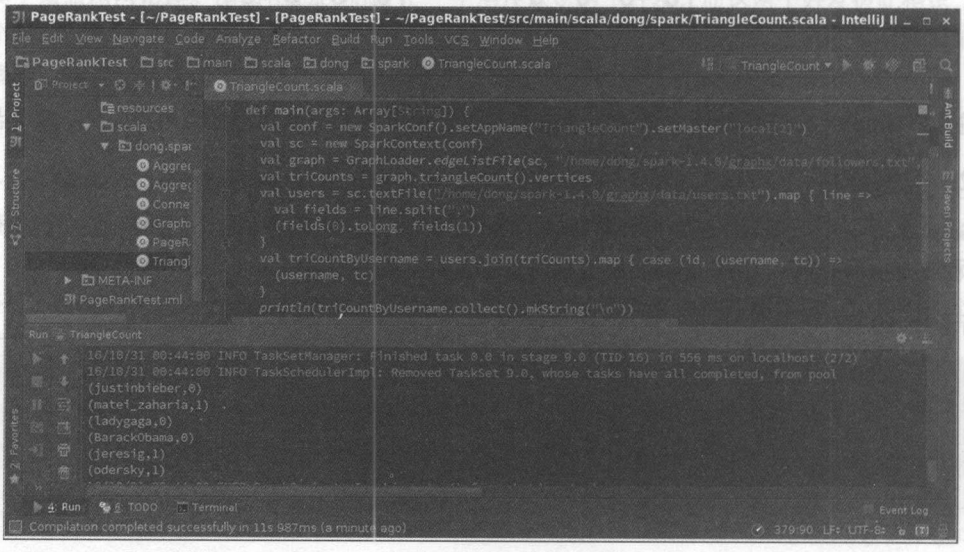


图 5.17　TriangleCount 运行结果示意图

第6章

Spark SQL

6.1 Spark SQL概述

Spark SQL 是 Spark 框架上用来处理结构化和半结构化数据的接口。Spark 早期为了提高 SQL-on-Hadoop 的效率，提出了一种分布式交互查询工具 Shark。相对于 Hive，Shark 使得 SQL 查询的速度提升了 10~100 倍，图 6.1 是在 100 个 EC2 节点上两者针对 1.7TB 数据的查询性能对比。然而随着 Spark 的发展，Shark 对 Hive 的诸多依赖（Hive 的 SQL 语法解析器、逻辑计算算子和逻辑计划树等）制约了 Spark 的交互式查询性能和各组件之间的相互集成，为此 Spark 提出了 Spark SQL。

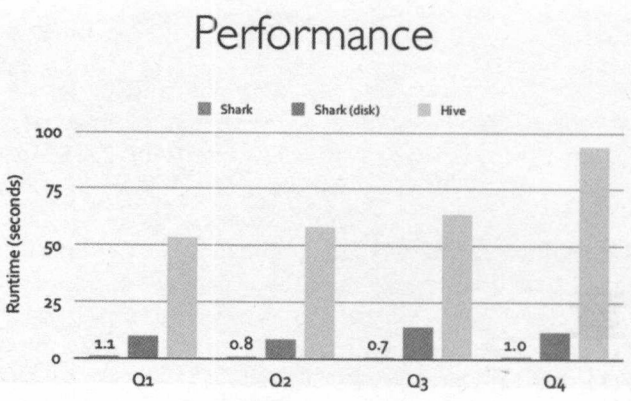


图 6.1　Hive 与 Shark 性能对比

Spark SQL 是 Spark 1.0 中提出的一个核心组件。Spark SQL 在汲取了 Shark 一系列优点的同时，不仅覆盖了 Shark 的全部功能，而且对其进行了扩展。由于摆脱了对 Hive 的依赖，Spark SQL 在查询性能、数据兼容、组件集成与扩展等方面都得到了极大的提升，如图 6.2 所示。用户借助于 Spark SQL 可以方便、高效地构建 Spark 大数据平台上的数据仓库，为 Spark 带来了通用、高效、多元一体的结构化数据处理能力。

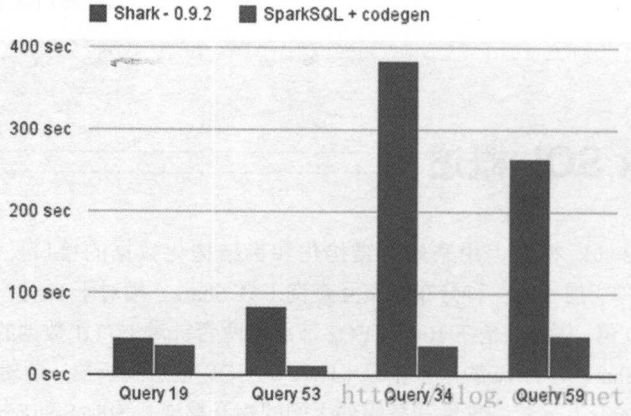


图 6.2　Spark SQL 与 Shark 性能对比

Spark SQL 相对于 Shark 在性能方面得到大幅提升主要基于以下几点优化。

1. 内存列存储

Spark SQL 的表数据在内存中存储不是采用原生态的 JVM 对象存储方式，而是采用内存列存储方式。该存储方式无论在空间占用量还是读取吞吐率上都占有很大优势。

2. 字节码生成技术

Spark 1.1.0 增加了 codegen 模块，使用动态字节码生成技术对匹配的表达式采用特定的代码动态编译。

3. Scala 代码优化

Spark SQL 在使用 Scala 编写代码的时候，能够尽量避免低效的、容易引起 GC（垃圾回收）的代码。

Spark SQL 作为 Spark 平台上的交互式查询工具，主要具有如下特点：

1. 面向多种语言（Java、Scala、Python 和 R），能够将 SQL 语句集成到 Spark 应用程序中，如图 6.3 所示。

```
context = HiveContext(sc)
results = context.sql(
  "SELECT * FROM people")
names = results.map(lambda p: p.name)
```

图 6.3 在 Spark SQL 查询结果上应用 Spark 中的 API

2. 具有统一的数据访问方式，不仅兼容 Hive，还可以从各种结构化数据源（例如 JSON、Hive、Parquet 等）中读取数据，如图 6.4 所示。

```
context.jsonFile("s3n://...")
  .registerTempTable("json")
results = context.sql(
  """SELECT *
     FROM people
     JOIN json ...""")
```

图 6.4 不同数据源的查询与连接

3. 不仅能与 Spark 上的其他组件无缝衔接，还支持从类似商业智能软件（例如 Tableau 等）的外部工具中通过标准数据库连接器（JDBC/ODBC）连接 Spark SQL 进行查询，如图 6.5 所示。

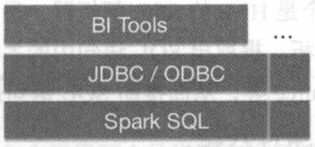


图 6.5. Spark SQL 结合商业智能软件的海量数据查询

4. 重用了 Hive 的前端与 MetaStore（元数据存储），完全兼容 Hive 数据的查询与 UDFs，如图 6.6 所示。

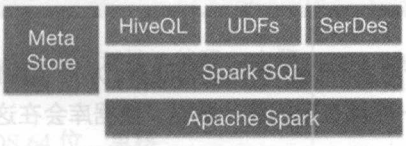


图 6.6 Spark SQL 兼容 Hive

5. 比 Shark 快了几十倍，具有更高效的分布式交互查询性能。

6.2 Spark SQL 逻辑架构

6.2.1 SQL 执行流程

与关系型数据库类似，Spark SQL 执行的 SQL 语句由 Projection、Data Source 和 Filter 组成，分别对应查询过程中的 Result、Data Source 和 Operation。Spark SQL 执行一个 SQL 语句的完整过程通常涉及以下 4 个步骤：解析（Parse）、绑定（Bind）、优化（Optimize）和执行（Execute），如图 6.7 所示。

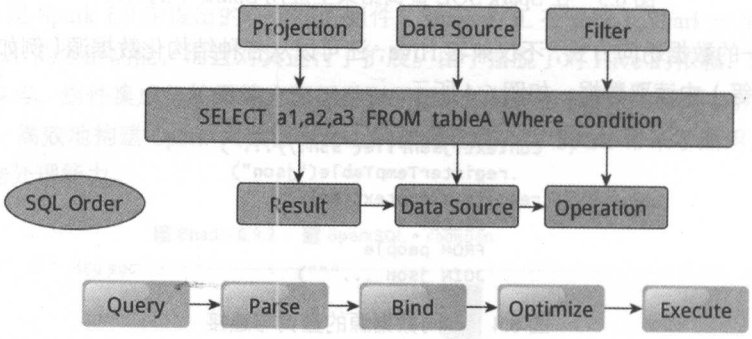


图 6.7 Spark SQL 执行流程

1. SQL 解析

Spark SQL 有两个解析器：一个是 Hive 的 SQL 解析器，另一个是内置的简单 SQL 解析器。解析器对 SQL 语句的语法结构进行解析，提取出 SQL 语句中的关键词（如 SELECT、FROM、Where 等）、表达式、Projection 和 Data Source 等，从而推断 SQL 语句是否规范。

2. 逻辑计划元数据绑定和语义分析

这个阶段需要参照数据表的元数据信息，主要涉及表中各列的名称、数据类型等。在此基础上将 SQL 语句和数据库的数据字典进行绑定，如果 Projection、Data Source 等描述的内容均存在且符合相关的类型约束，则表示该 SQL 语句是可以执行的。

3. 逻辑计划优化

Spark SQL 以规则的方式针对各种优化条件进行映射，从而实现优化逻辑计划。一般的数据库会提供几个执行计划，这些计划一般都有运行统计数据，数据库会在这些计划中选择一个最优计划。

4．物理计划生成与执行

Spark SQL 基于规则选取不同的物理执行算子把优化后的逻辑计划转换成物理计划，并最终提交给执行引擎以 Operation→Data Source→Result 的次序执行物理计划。在执行过程中，有时甚至不需要读取物理表就可以返回结果，例如重复执行刚提交运行过的 SQL 语句，可直接从数据库的缓冲池中获取查询结果。

6.2.2 Catalyst

Spark SQL 在对 SQL 语句处理时，首先将 SQL 语句进行解析，然后形成一棵逻辑计划树；采用规则的形式，通过模式匹配对树中不同类型的节点执行不同的操作。在整个处理过程中，树和规则相互配合，进而完成解析、绑定、优化和执行步骤。而上述这 4 个步骤的实现与优化均封装在一个可扩展的模块 Catalyst 中，因此 Catalyst 是决定 Spark SQL 性能的关键。

在解析阶段，无论何种 SQL 解析器，最终都映射成 Catalyst 的逻辑计划树，针对不同的 SQL 解析器，生成不同的映射代码。在绑定阶段，Catalyst 实现并扩展了对元数据的管理，通过规则进行语义分析和绑定，相对于 Shark 完全复用了 Hive 的语义分析库，Catalyst 的元数据管理默认有两个扩展，分别是完全基于内存的简单元数据管理和基于 Hive 的 MetaStore。在优化阶段，相对于 Shark 完全复用 Hive 的逻辑优化模块，Catalyst 完全基于 Scala 重写了逻辑计划优化框架，以规则的方式对各种优化条件进行映射，提出一套独立的规则集对逻辑计划树进行优化，进而达到优化逻辑计划的目的。在执行阶段，Catalyst 提供了 Spark 策略集合，利用该集合中的规则选取不同的操作，把优化后的逻辑计划变成物理计划，并最终提交执行引擎运行。

Catalyst 用 Context（包括 HiveContext 和 SQLContext）把不同阶段的功能模块串起来。其中 SQLContext 支持 SQL 语法解析器；HiveContext 支持 SQL 语法解析器和 hiveSQL 语法解析器，默认为 hiveSQL 语法解析器，用户可以通过配置切换成 SQL 语法解析器来运行 hiveSQL 不支持的语法。

6.3 Spark SQL CLI

CLI（Command-Line Interface，命令行界面）是指可在提示符下输入可执行指令的界面，用户通过键盘输入指令，计算机接收到指令后予以执行。Spark SQL CLI 是指使用命令行界面直接输入 SQL 指令，然后提交给 Spark 集群执行，并在界面中显示运行过程和最终的结果。

6.3.1 硬软件环境

1. 主机操作系统：Windows 64 位，双核 4 线程，主频 2.2G，10G 内存
2. 虚拟软件：VMware
3. 虚拟机操作系统：CentOS 64 位，单核
4. 虚拟机运行环境

JDK：1.7.0_55（64 位）

Hadoop：2.2.0

Scala：2.10.4

Spark：1.4.0

Hive：0.13.1

6.3.2 集群环境

集群包含三个节点，节点之间可以免密码 SSH 访问，节点 IP 地址和主机名分布如表 6-1 所示。

表 6-1 集群环境

序　号	IP 地址	机器名	类　型	核数/内存	用户名
1	192.168.149.132	master	NN/RM/Master	1 核/2G	dong
2	192.168.149.133	slave1	DN/NM/Worker	1 核/2G	dong
3	192.168.149.134	slave2	DN/NM/Worker	1 核/2G	dong

6.3.3 结合 Hive

在运行 Spark SQL CLI 中需要使用 Hive 的 MetaStore，因此需要在 Spark 中添加其 uris。具体方法是在 SPARK_HOME/conf 目录下创建 hive-site.xml 文件，然后在该配置文件中添加 hive.metastore.uris 属性，具体如下：

```
    <configuration>
    <property>
    <name>hive.metastore.uris</name>
    <value>thrift://master:9083</value>
    <description>Thrift URI for the remote metastore. Used by metastore client
to connect to remote metastore.
    </description>
    </property>
    </configuration>
```

6.3.4 启动 Hive

在使用 Spark SQL CLI 之前需要启动 Hive 的 MetaStore。如果数据存放在 HDFS 文件系统中，还需要启动 Hadoop 的 HDFS。使用下面的命令可启动 Hive 的 MetaStore 并在后台运行。

```
    $nohup hive --service metastore > metastore.log 2>&1 &
```

6.4 DataFrame 编程模型

6.4.1 DataFrame 简介

相对于传统的 MapReduce API，Spark 通过 RDD 和函数式编程模式把分布式数据处理转换成分布式数据集的处理，提供了更高抽象层次的、更丰富的 API，极大地简化了并行程序的开发过程，提高了开发效率。然而，对于没有 MapReduce 和函数式编程经验的新手来说，熟练掌握 Spark 中的 API 仍然存在着一定的门槛。另一方面，数据科学家们所熟悉的 R、Pandas 等传统数据处理工具虽然提供了直观的 API，但却局限于单机处理，无法胜任大数据场景。为了降低 Spark 的使用门槛，推进其广泛应用，Spark 1.3.0 引入了 DataFrames。DataFrames 同时支持 Scala、Java 与 Python 等多种语言，方便更多的数据分析师基于 Spark 开展数据分析工作。

DataFrame 是一个以命名列方式组织的分布式数据集，与传统的关系型数据库中的表结构类似。与一般的 RDD 不同，DataFrame 带有结构信息（即数据字段），即 DataFrame 所表示的表数据集的每一列都带有名称和类型，对于数据的内部结构具有很强的描述能力。因此 Spark SQL 可以对隐藏在 DataFrame 背后的数据源以及作用于 DataFrame 之上的变换进行有针对性的优化，从而大幅提升 SQL 查询的效率。

在 Spark 1.3.0 版本之前，Spark SQL 使用 SchemaRDD 表示和存储结构化的数据，并借此实现对数据的一系列操作。SchemaRDD 是存放 Row 对象的 RDD，每个 Row 对象代表一行记录。SchemaRDD 借助于记录中的结构信息实现了高效的数据存储。例如对于包含 Name、Age、Height 三个成员属性的 Person 类型数据集，分别采用 RDD 和 DataFrame 进行表示，逻辑框架如图 6.9 所示。区别于先前版本中的 SchemaRDD，DataFrame 不再直接自 RDD 继承，但可以通过相关方法实现到 RDD 的转换。DataFrame 对 SchemaRDD 的功能进行了扩展，更加适用于处理分布式大数据场景。分别基于 RDD 与 DataFrame 对 1000 万个整数对进行聚合操作，性能对比如图 6.10 所示。

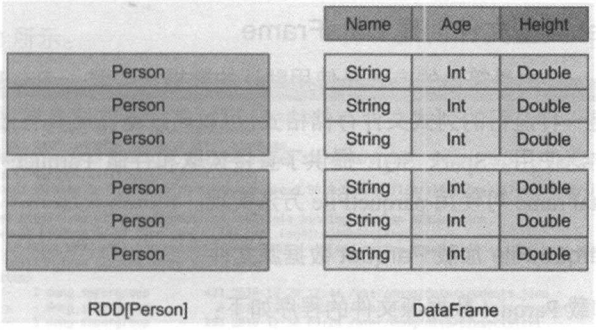


图 6.9 基于 RDD（左图）与 DateFrame（右图）表示与存储的 Person 类型数据集

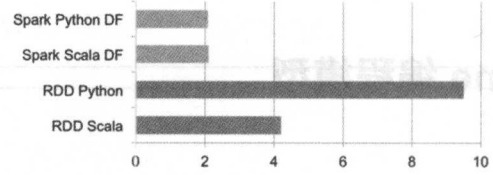


图 6.10　包含 1000 万个整数对的 RDD 与 DataFrame 聚合操作性能对比

DataFrames 主要特点如下：

1. 支持单机 KB 级到集群 PB 级的数据处理。
2. 支持多种数据格式和存储系统。
3. 通过 Spark SQL Catalyst 优化器可以进行高效的代码生成与优化。
4. 能够无缝集成 Spark 上的大数据处理工具。
5. 提供了 Python、Java、Scala、R 多种语言的 API。

6.4.2　创建 DataFrames

在实际应用场景中通常会涉及多种类型的数据源。Spark SQL 不仅支持从多种不同类型数据源加载文件直接创建 DataFrame，还支持将 DataFrame 保存为多种不同类型的数据文件，如图 6.11 所示。本节将重点介绍基于 Parquet 与 JSON 类型文件创建 DataFrame，而基于 Hive 表中数据创建 DataFrame 将在后面章节以案例形式介绍。

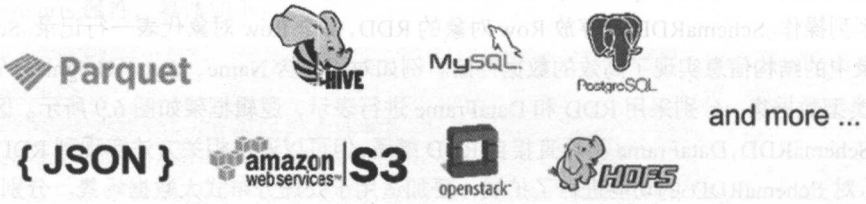


图 6.11　Spark SQL 支持的数据源类型

1. 基于 Parquet 类型文件创建 DataFrame

加载与保存 DataFrame 的最简单的方式是使用默认的数据源类型， Spark SQL 默认的数据源类型为 Parquet。Parquet 是一种流行的列式文件存储格式，可以高效地存储具有嵌套字段的记录。Parquet 格式在 Hadoop 中被广泛使用。Spark SQL 提供了直接读取和存储 Parquet 格式文件的方法。基于 Parquet 数据源创建 DataFrame 可使用 parquetFile 方法实现。

示例 6-1　采用 parquetFile 加载 Parquet 数据源文件

采用 parquetfiee 加载 Parquet 数据源文件的程序如下：

```
    scala>val
stu_parquet1=sqlContext.parquetFile("/user/dong/data/students.parquet")
```

执行上述代码后，students 即为创建出的一个 DataFrame。也可采用 load 方法，同样会得到一个 DataFrame。

运行结果如下：

```
stu_parquet1: org.apache.spark.sql.DataFrame = [id: string, name: string, age: int]
```

Spark SQL 测试数据目录示意图如图 6.12 所示。

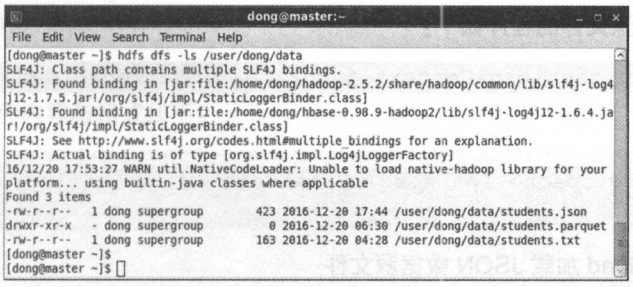


图 6.12 Spark SQL 测试数据目录示意图

示例 6-2 采用 load 加载 Parquet 数据源文件

采用 load 加载 Parquet 数据源文件的程序如下：

```
scala>val stu_parquet2 = sqlContext.load("/user/dong/data/students.parquet", "parquet")
```

运行结果如下：

```
stu_parquet2: org.apache.spark.sql.DataFrame = [id: string, name: string, age: int]
```

示例 6-3 保存 Parquet 数据源文件

```
scala>stu_parquet2.save("/user/dong/data/students_copy.parquet")
```

运行结果如图 6.13 所示。

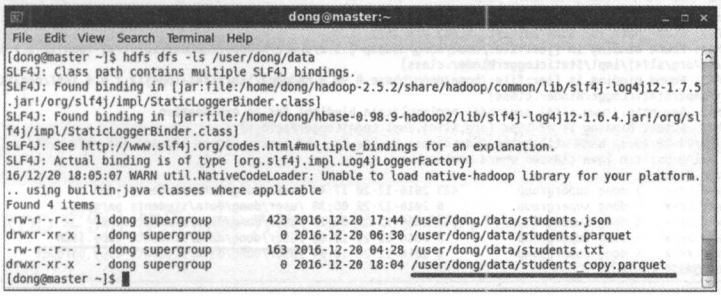


图 6.13 示例 6-3 运行结果示意图

2. 基于 JSON 类型文件创建 DataFrame

若一个 JSON 文件中的数据遵循相同的结构信息，那么 Spark SQL 可以通过扫描该文件推测出该结构信息，并允许用户直接使用列名访问列下的各项数据。对于一个包含大量 JSON 文件的目录，Spark SQL 的结构信息推断功能能够让用户非常高效地对结构化的数据进行操作，而无须编写专门的代码来读取不同结构的文件。基于 JSON 数据源创建 DataFrame 可使用 jsonFile 方法实现。

示例 6-4　加载 JSON 数据源文件

加载 JSON 数据源文件的程序如下：

```
scala>val stu_json1=sqlContext.jsonFile("/user/dong/data/students.json")
```

运行结果如下：

```
stu_json1: org.apache.spark.sql.DataFrame = [age: string, id: string, name: string]
```

示例 6-5　采用 load 加载 JSON 数据源文件

采用 load 加载 JSON 数据源文件的程序如下：

```
scala>val stu_json2=sqlContext.load("/user/dong/data/students.json", "json")
```

运行结果如下：

```
stu_json2: org.apache.spark.sql.DataFrame = [age: string, id: string, name: string]
```

示例 6-6　保存 JSON 数据源文件

保存 JSON 数据源文件的程序如下：

```
scala>stu_json2.save("/user/dong/data/students_copy.json")
```

运行结果如图 6.14 所示。

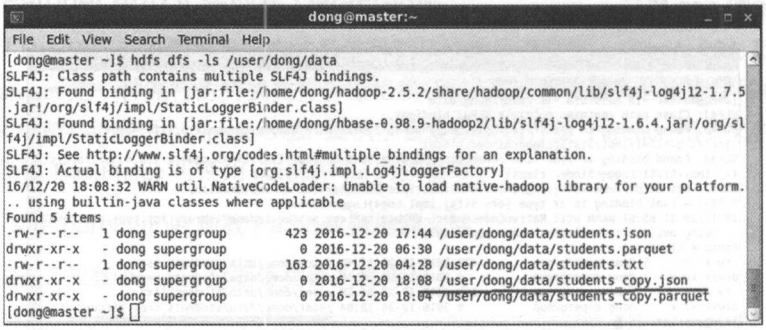


图 6.14　示例 6-6 运行结果示意图

3. 基于 RDD 创建 DataFrame

Spark SQL 支持两种方法将存在的 RDD 转换为 DataFrame。

（1）通过反射机制创建 DataFrame

Spark SQL 的 Scala 接口支持将包含样本类的 RDD 自动转换为 DataFrame。首先通过样本类定义数据的模式；然后 Spark SQL 通过反射读出样本类中的参数名称，并作为表中字段的名称，样本类可以嵌套或者包含复杂的类型（如序列、数组等）；最后注册为一个表以供后续的 SQL 查询使用。

示例 6-7 反射机制创建 DataFrame

第 1 步，以 students.txt 作为测试数据。

students.txt 中文件内容与格式如下：

```
160201,Michael,17
160101,Andy,23
160301,justin,23
160202,John,22
160102,Herry,17
160203,Brewster,18
160302,Brice,20
160303,Justin,25
160103,Jerry,22
160304,Tom,24
```

将 students.txt 上传至 HDFS，上传后对应目录如图 6.14 所示。

第 2 步，定义 students.txt 对应的样本类 Student。

```
case class Student(id: String, name : String, age: Int)
```

第 3 步，加载文件并转换为 RDD[Student]。

通过 SparkContext 的 textFile 方法逐行加载文件中的数据；然后以“,”作为分隔符，将每一行字符串切分为一个只包含两个元素（分别代 name 和 age）的字符串数组；继续使用 map 方法将数组映射成样本类 Student 的实例，进而得到该文件对应的 RDD，其中元素类型为 Students。

```
scala>val rdd=sc.textFile("/user/dong/data/students.txt").map(_.split
(",")).map(p => Student(p(0), p(1), p(2).trim.toInt))
```

运行结果如下：

```
rdd: org.apache.spark.rdd.RDD[Student] = MapPartitionsRDD[25] at map at
<console>:23
```

第 4 步，在 RDD 实例上使用 toDF 方法，转换为对应的 DataFrame。

```
scala>val students = rdd.toDF()
```

运行结果如下：

```
students: org.apache.spark.sql.DataFrame = [id: string, name: string, age: int]
```

第 5 步，将 DataFrame 注册成临时表。

```
students.registerTempTable("tb_students")
```

第 6 步，基于临时表执行 SQL 语句。

注册成临时表后就可以使用 sqlContext 支持的 SQL 语句对该表进行查询操作。

```
scala>val youngstudents=sqlContext.sql("SELECT name FROM tb_students WHERE age>=19 AND age<=22")
```

运行结果如下：

```
youngstudents: org.apache.spark.sql.DataFrame = [name: string]
```

第 7 步，查看查询结果。

```
youngstudents.show
```

运行结果如图 6.15 所示。

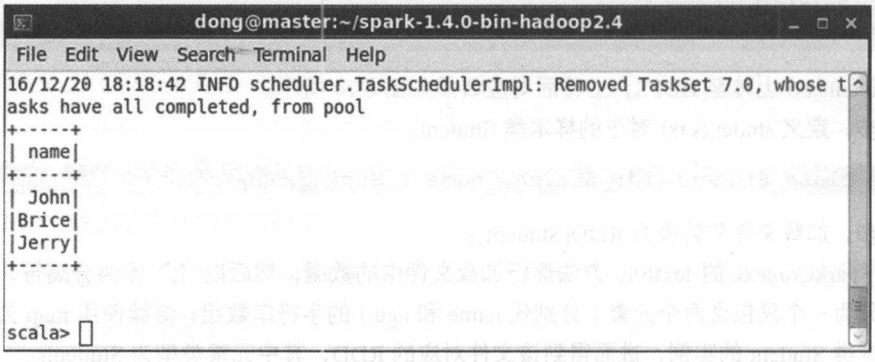


图 6.15　示例 6-7 运行结果示意图

第 6 步中使用 SELECT 语句从表“students”中查找出年龄在 19 到 22 之间的学生的名字。执行 SQL 语句后得到的结果是 DataFrame 实例，其支持 RDD 的全部操作。

需要注意的是，在以 spark-submit 方式提交的应用程序中，不要将样本类 Student 定义在 SparkContext 实例的作用域中，即需要将样本类定义在 main 方法外或 Object 外。

（2）自定义转换接口

虽然这种方法代码较为冗长，但是它允许用户在不知道列以及列的类型的情况下基于已有的 RDD 构造 DataFrame。

示例 6-8 自定义转换接口创建 DataFrame

第 1 步，导入相关包。

```
scala> import org.apache.spark.sql.types._
scala>import org.apache.spark.sql._
```

第 2 步，基于 HDFS 上的文本文件创建 RDD。

```
scala>val students=sc.textFile("/user/dong/data/students.txt")
```

运行结果如下：

```
students: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[31] at
textFile at <console>:24
```

第 3 步，定义数据文件中涉及的模式信息。

```
scala>val schemaString="id name age"
```

运行结果如下：

```
schemaString: String = id name age
```

第 4 步，创建一个由 StructType 表示的数据模式，指定每一列都为 StringType 类型，且可为 null。

```
val schema=StructType(schemaString.split(" ").map(fieldname=> StructField
(fieldname, StringType, true)))
```

运行结果如下：

```
schema: org.apache.spark.sql.types.StructType = StructType(StructField(id,
StringType,true), StructField(name,StringType,true),StructField(age,StringType,
true))
```

第 5 步，将 RDD 的字符串元素转换为 DataFrame 的 Row 类型。

```
val rowRDD=students.map(_.split(",")).map(p=>Row(p(0), p(1), p(2).trim))
```

运行结果如下：

```
rowRDD: org.apache.spark.rdd.RDD[org.apache.spark.sql.Row] =
MapPartitionsRDD[33] at map at <console>:29
```

第 6 步，利用第 4 步创建的数据模式创建 DataFrame。

```
val studentsDataFrame=sqlContext.createDataFrame(rowRDD, schema)
```

运行结果如下：

```
studentsDataFrame: org.apache.spark.sql.DataFrame = [id: string, name:
string, age: string]
```

第 7 步，将创建出的 DataFrame 注册到临时表中。

```
scala>studentsDataFrame.registerTempTable("tb_students")
```

第 8 步：基于临时表执行 SQL 语句。

```
scala>val names=sqlContext.sql("SELECT name FROM tb_students")
```

第 9 步：查看查询结果。

```
names.show
```

运行结果如图 6.16 所示。

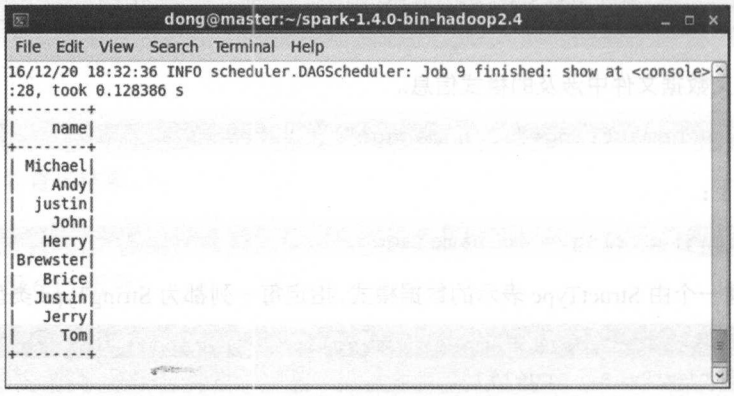


图 6.16　示例 6-8 运行结果示意图

6.4.3　保存 DataFrames

1．保存为临时表

使用 registerTempTable 可创建一个临时表，并将 DataFrame 保存至该表中。临时表的生命周期包含于其所在的 sqlContext 或 hiveContext 实例中，即在应用退出后临时表会自动销毁。需要注意，一个 sqlContext（或 hiveContext）中的临时表不能同时被另一个 sqlContext（或 hiveContext）使用。

示例 6-9

```
    scala  >val  students=sqlContext.load("/user/dong/data/students.parquet",
"parquet")
    scala >students.registerTempTable("tb_students")
```

上述示例首先加载文件 students.parquet 中的数据创建 DataFrame 得到 students，然后利用 registerTempTable 方法将 students 保存到表 tb_students 中。

2．保存为持久化表

相对于 registerTempTable，采用 saveAsTable 方法将数据以表的形式持久化到存储系统上，因此

只要连接存在，Spark 重启后表中数据仍然存在。

示例 6-10

```
scala>val students=sqlContext.load("/user/dong/data/students.json", "json")
scala>students.saveAsTable("tb_students")
```

加载文件 students.json 中的数据创建 DataFrame 得到 students，然后利用 saveAsTable 方法将 students 保存到表 tb_students 中。

3. 缓存表

缓存表是将数据存储到内存中，是 Spark SQL 性能调优的一种重要方式。Spark SQL 可以通过 sqlContext 调用 cacheTable 方法将表以列优先的形式存储到内存。当 Spark 进行相关 SQL 查询时，将会仅仅浏览需要的列并且自动地压缩数据，以减少内存的使用以及垃圾回收的压力。可以通过 sqlContext 调用 uncacheTable 方法删除缓存在内存中的表。需要注意，如果使用先前版本中的 schemaRDD 调用 cache 方法，表将不会以列优先形式存储。为了提高 SQL 查询效率，推荐使用 DataFrame 调用 cacheTable 方法。

示例 6-11

使用 cacheTable 缓存表。

```
scala>sqlContext.cacheTable("tb_students")
```

使用 uncacheTable 删除缓存表。

```
scala>sqlContext.uncacheTable("tb_students")
```

6.5 DataFrame 常见操作

Spark SQL 针对 DataFrame 提供了丰富的 API，前面章节中已经介绍了一些相关操作，大多与 RDD 中的相关操作功能类似，例如 map、count、take 等，针对这些操作本节不再赘述。

6.5.1 数据展示

DataFrame 的 collect、take、first 方法用于将数据展示在界面上，其语法及输出效果与 RDD 的同名方法类似。此外，DataFrame 还定义了 head 和 show 方法用于展示数据。

1. head(num)

返回 DataFrame 中的前 num 行数据，若不指定参数 num 则仅返回第一行。

示例 6-12

```
scala>students.head(3)
```

运行结果如下。

```
res16:  Array[org.apache.spark.sql.Row]  =  Array([17,160201,Michael],
[23,160101,Andy], [23,160301,justin])
```

2. show(num)

show 方法的功能与 head 类似，但该方法提供了一种可读性更好的展示方式。show 方法返回数据的前 num 行，并可以选择将较长的字符串进行截断展示，默认为 20 行。

示例 6-13

```
scala>students.show(3)
```

运行结果如图 6.17 所示。由此可见，show 方法的展示形式更符合人们对于“表”的认知。

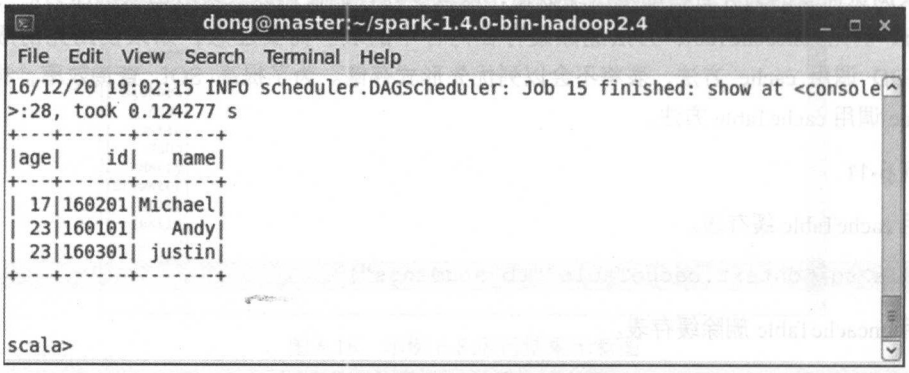


图 6.17　示例 6-13 运行结果示意图

6.5.2　常用列操作

常用的列操作包括选择列、删除列、重命名列、查看和修改列的类型等。

1. columns

以数组形式返回 DataFrame 中的全部列名。

示例 6-14

```
scala>students.columns
```

运行结果：

```
res19: Array[String] = Array(age, id, name)
```

2. dtypes

以二元组数组的形式返回 DataFrame 中的全部列名及其对应的类型。

示例 6-15

```
scala>students.dtypes
```

运行结果如下。

```
res20: Array[(String, String)] = Array((age,StringType), (id,StringType),
(name,StringType))
```

3. printSchema

与 dtypes 方法功能类似，printSchema 以树形结构返回 DataFrame 中的 Schema 信息。

示例 6-16

```
scala>students.printSchema
```

运行结果如图 6.18 所示。

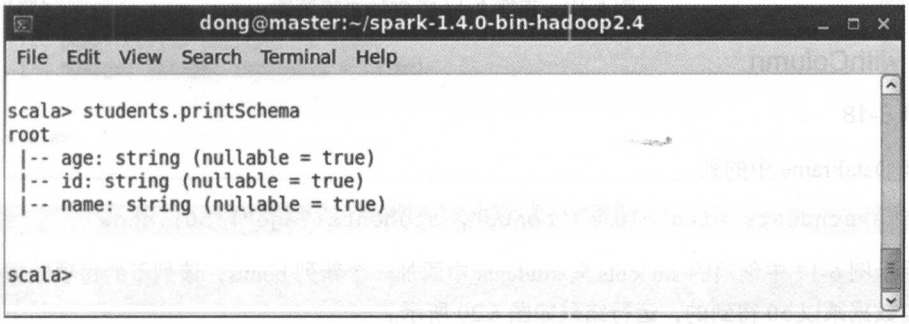


图 6.18 示例 6-16 运行结果示意图

4. select

从当前 DataFrame 中选取指定的列数据创建并返回新的 DataFrame。select 提供了三种指定列的方式，可以采用列名字符串的重复参数、Column 重复参数、列名表达式的重复参数。

示例 6-17

```
scala> case class Student(id: String, name : String, age: Int)
scala> val
rdd=sc.textFile("/user/dong/data/students.txt").map(_.split (",")).map(p =>
Student(p(0), p(1), p(2).trim.toInt))
scala> val students = rdd.toDF()
scala>students.select("age", "name").show
scala>students.select( $"age", $"name").show
```

上述代码首先创建样例类 Student；然后基于文本文件创建 RDD；再利用 toDF 方法创建 DataFrame 即 students；最后通过 select 方法查看 students 中的 age 与 name 信息。运行结果如图 6.19 所示。

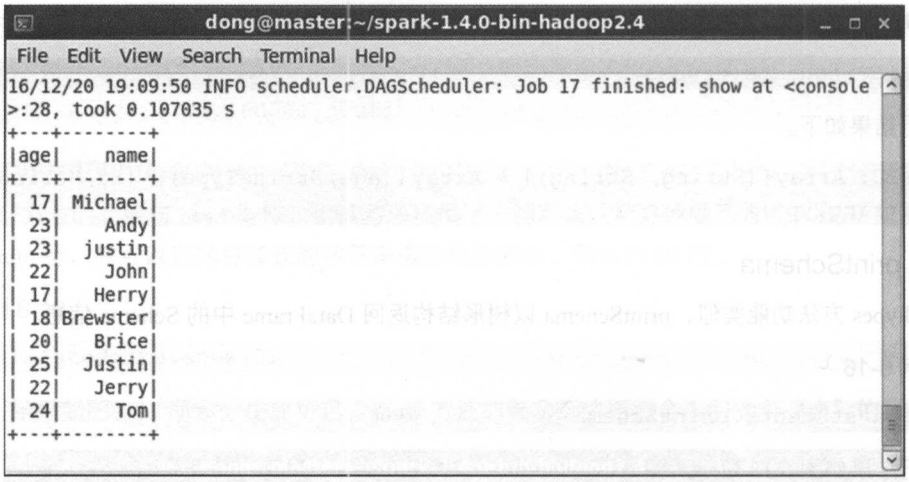


图 6.19　示例 6-17 运行结果示意图

5. withColumn

示例 6-18

增加 DataFrame 中的列。

```
scala>students.withColumn("bonus", students("age")*50).show
```

沿用示例 6-17 中创建的 students 向 students 中添加一个新列 bonus，该列下的每项数据是由 age 列中相应数据乘以 50 得到的，运行结果如图 6.20 所示。

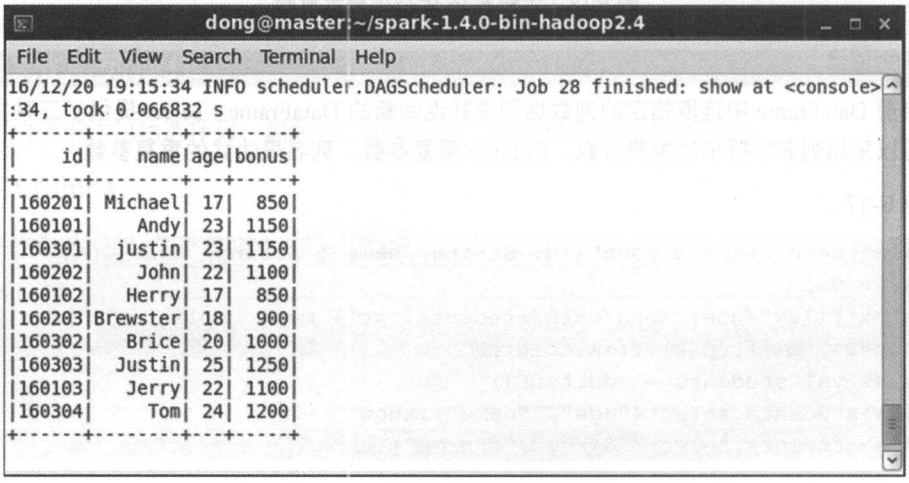


图 6.20　示例 6-18 运行结果示意图

6. withColumnRenamed(existing, new)

对 DataFrame 中指定的列重命名，其中参数 existing 是当前列名称；参数 new 是新列名称，二者类型均为字符串。

示例 6-19

```
scala>val newstudents=students.withColumnRenamed("age", "newage")
scala>newstudents.printSchema
```

沿用示例 6-17 中创建的 students，将 students 中的列 age 重命名为 newage，运行结果如图 6.21 所示。

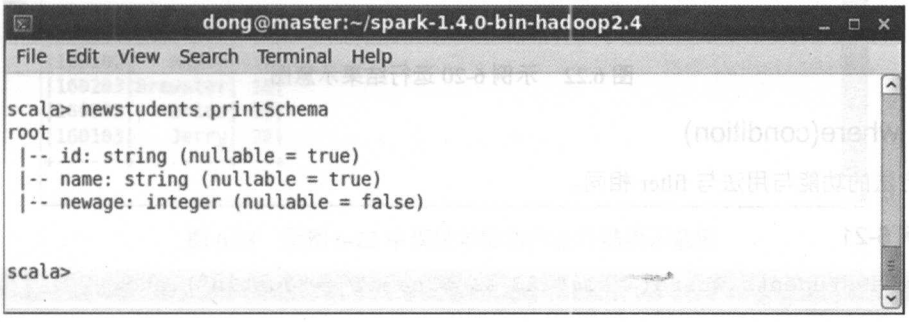


图 6.21 示例 6-19 运行结果示意图

6.5.3 过滤

在很多场景中需要按一定的条件对数据进行筛选过滤，为此 DataFrame 提供了 filter、where 操作，二者功能相同，互为别名，其用法与 RDD 的 filter 方法类似。

1. filter(condition)

该方法按指定条件对数据进行过滤并返回新的 DataFrame，其中参数 condition 为列的表达式或是字符串形式的 SQL 语句。

示例 6-20

```
scala> students.filter($"age">20).show
```

沿用示例 6-17 中创建的 students，筛选出年龄大于 20 岁的学生。运行结果如图 6.22 所示。

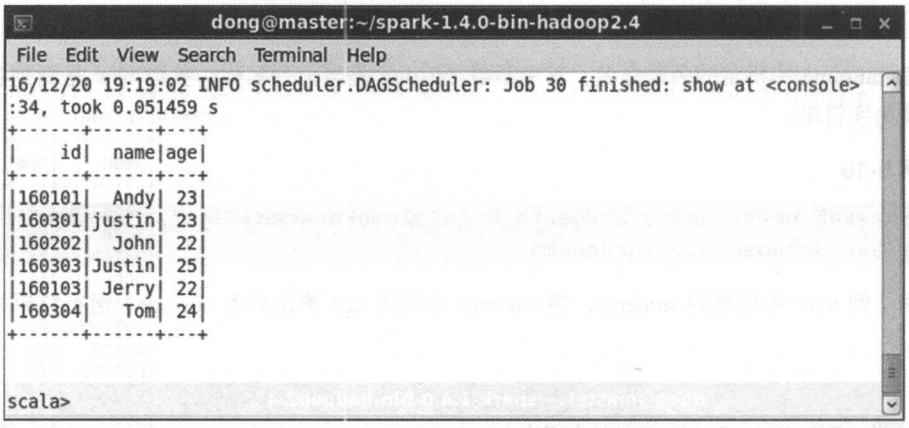


图 6.22　示例 6-20 运行结果示意图

2. where(condition)

该方法的功能与用法与 filter 相同。

示例 6-21

```
scala>students.where($"age">23 && $"name"==="Justin").show
```

沿用示例 6-17 中创建的 students，查看 students 中年龄大于 23 岁且姓名为 Justin 的记录，运行结果如图 6.23 所示。

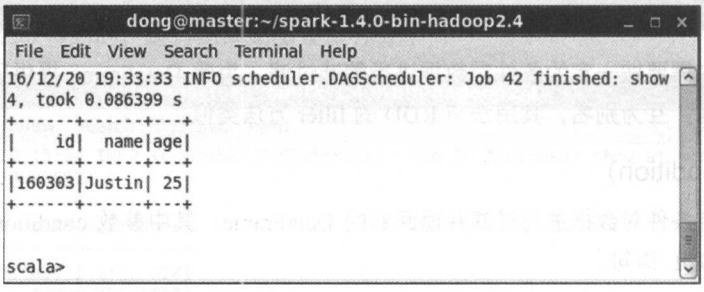


图 6.23　示例 6-21 运行结果示意图

6.5.4　排序

在进行数据分析时，往往需要按指定的列进行排序（例如按时间、编号等进行排序）。排序后的数据不仅方便用户查阅，而且有助于提高后续很多其他操作的执行效率。

1. orderBy(cols*)

按照指定的一列或多列进行排序，其中参数 cols 是列名列表，以字符串形式或列对象的形式传入，orderBy 可以接收多个参数。

示例 6-22

```
scala> students.orderBy(students("age")).show(5)
```

沿用示例 6-17 中创建的 students，对 students 中的记录仅按照年龄排序，默认为升序排序，运行结果如图 6.24 所示。

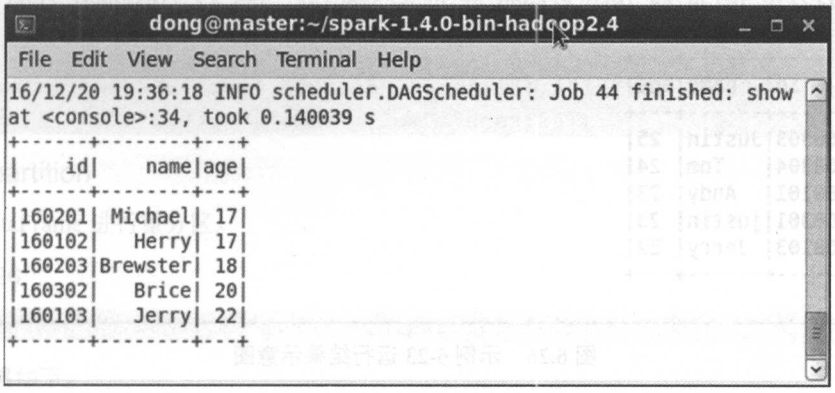


图 6.24 示例 6-22 中仅按年龄排序运行结果示意图

```
scala> students.orderBy("age","id").show(5)
```

对 students 中的记录先按照年龄排序，再按照 id 排序，默认均为升序排序，运行结果如图 6.25 所示。注意图 6.24 与图 6.25 中前两条记录，Herry 与 Michael 同为 17 岁，但 Herry 的 id 号小于 Michael 的 id 号，因此若再按 id 默认排序，Herry 对应的记录在前。

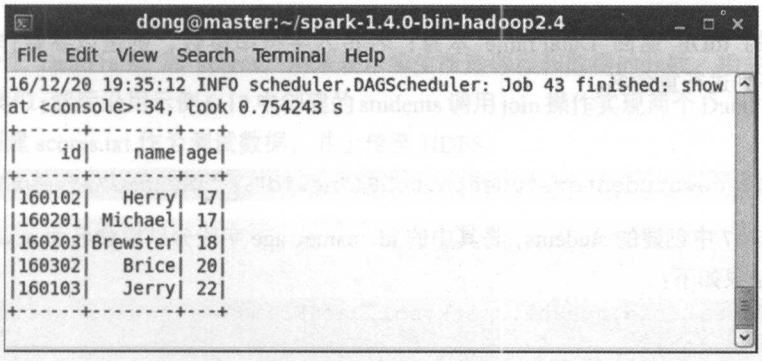


图 6.25 示例 6-22 中按年龄与 id 排序运行结果示意图

2. sort(cols*)

该方法按照指定的一列或多列进行排序。

示例 6-23

```
scala>students.sort($"age".desc).show(5)
```

沿用示例 6-17 中创建的 students，对 students 中的记录先按照年龄排序，desc 表示降序排序，运行结果如图 6.26 所示。

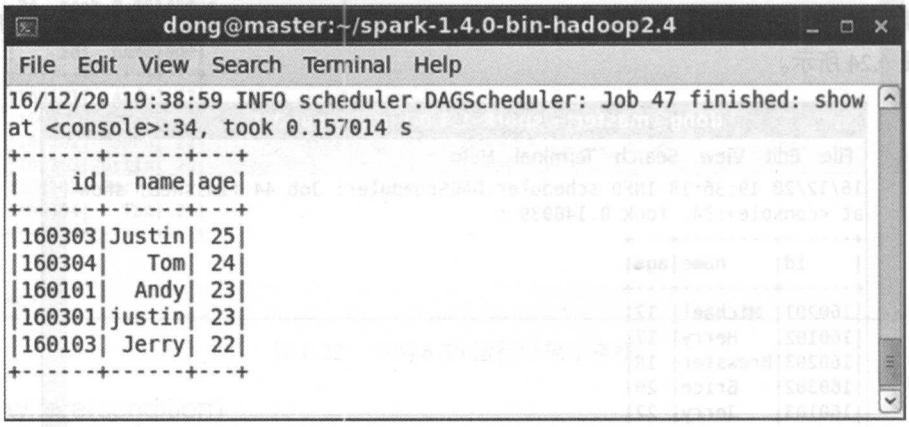


图 6.26　示例 6-23 运行结果示意图

需要注意，在定义数据类型时若 age 和 id 都是字符串，则对这两列执行排序运算是按照字符串排序而不是整数排序；如果对列执行的是算术运算，则 Spark 会通过类型推断先将列下的字符串转化为数值再进行计算。

6.5.5　其他常见操作

1．toDF

不带参数的 toDF 返回 DataFrame 本身；若传入字符串参数，则是以参数作为新列名对原 DataFrame 中各列进行重命名。

示例 6-24

```
scala>val newstudents=students.toDF("newid", "newname","newage")
```

沿用示例 6-17 中创建的 students，将其中的 id、name、age 字段分别重命名为 newid、newname、newage，运行结果如下：

```
newstudents: org.apache.spark.sql.DataFrame = [newid: string, newname: string, newid: int]
```

需要注意，调用带参的 toDF 方法时，参数个数必须和调用者 DataFrame 的列个数一样。

2．toJSON

把 DataFrame 的内容用包含 JSON 字符串的 RDD 返回。

示例 6-25

```
scala>students.toJSON.collect
```

运行结果如下：

```
    res48: Array[String] = Array({"id":"160201","name":"Michael","age":17},
{"id":"160101","name":"Andy","age":23},
{"id":"160301","name":"justin","age":23},
{"id":"160202","name":"John","age":22},
{"id":"160102","name":"Herry","age":17},
{"id":"160203","name":"Brewster","age":18},{"id":"160302","name":"Brice","ag
e":20},{"id":"160303","name":"Justin","age":25},{"id":"160103","name":"Jerry
","age":22}, {"id":"160304","name":"Tom","age":24})
```

3. repartition

对原 DataFrame 进行重分区。

示例 6-26

```
scala>students.repartition(4).rdd.partitions.size
```

运行结果如下。

```
res49: Int = 4
```

分区后返回的 DataFrame 对应的 RDD 分区个数已经改为 4。

4. join

连接两个 DataFrame，也可通过表达式参数给出连接的依据、连接方式，默认为 inner 方式。

示例 6-27

首先创建一个 DataFrame 即 scores，用来表示学生所选课程和取得的分数，相应的操作与示例 6-7 中步骤 1~4 类似；然后沿用示例 6-17 中创建的 students 调用 join 操作实现两个 DataFrame 的连接。

第 1 步，创建 scores.txt 作为测试数据，并上传至 HDFS。

scores.txt 中文件内容与格式如下：

```
160201,1,60
160101,2,90
160301,1,70
160202,3,70
160102,3,50
160102,4,95
160302,1,98
160303,2,57
160103,3,64
160304,3,50
160201,2,77
160101,3,57
```

```
160301,3,72
160202,2,80
160102,2,58
160102,3,97
160302,4,91
160303,1,67
160103,2,62
160304,4,71
```

上传该文件至 HDFS 上的相应目录中，如图 6.27 所示。

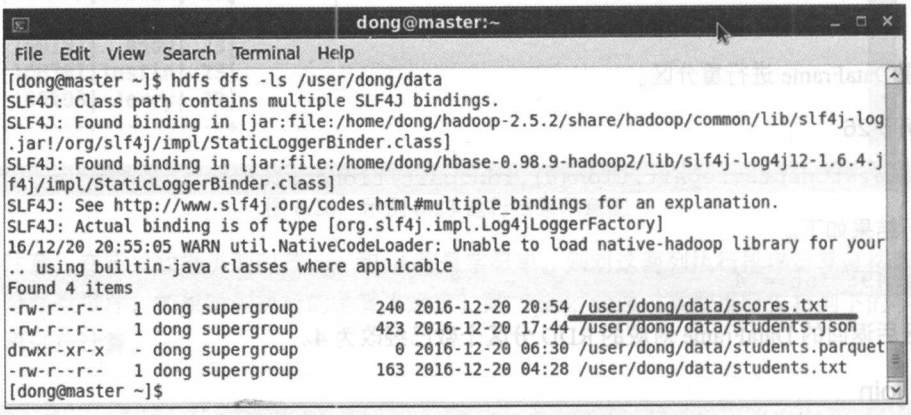


图 6.27　HDFS 上 sores.txt 所在目录示意图

第 2 步，定义 scores.txt 对应的样本类 Score。

```
case class Score(id:String,course_id:String,score:Int)
```

第 3 步，加载文件并转换为 RDD[Score]。

```
scala> val
scores_rdd=sc.textFile("/user/dong/data/scores.txt").map(_.split(",")).map(p
=> Score(p(0), p(1), p(2).trim.toInt))
```

第 4 步，在 RDD 实例上使用 toDF 方法，转换为对应的 DataFrame。

```
scala>val scores = scores_rdd.toDF()
```

第 5 步，调用 join 操作对两个 DataFrame 进行连接。

```
scala> students.join(scores, students("id" )===scores("id"), "outer").show
```

本例中通过"outer"参数指明连接方式为外连接，运行结果如图 6.28 所示。

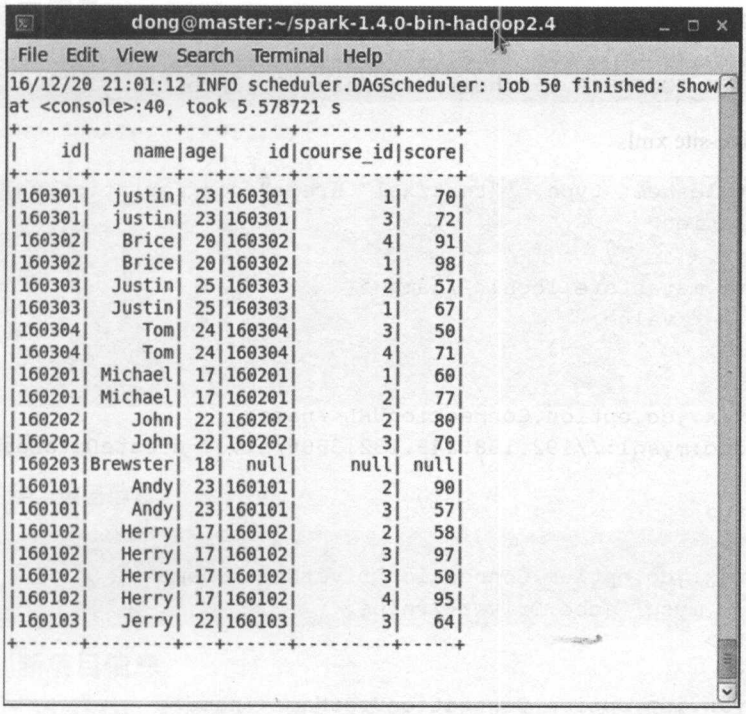


图 6.28 示例 6-27 运行结果示意图

6.6 基于 Hive 的学生信息管理系统的 SQL 查询案例与解析

实际应用场景中的离线数据分析任务大多是基于 Hive 实现的。Spark SQL 同样支持读写存储在 Hive 中的数据，这大大降低了 Hive 向 Spark SQL 中的迁移工作量。本节首先介绍 Spark SQL 与 Hive 的整合步骤，然后给出一个基于 Hive 的学生信息管理系统简单案例。

6.6.1 Spark SQL 整合 Hive

1. 安装 MySQL

首先安装 MySQL。

2. 安装 Hive 环境，当前 Hive 环境使用 MySQL 作为默认的元数据数据库

（1）将 mysql-connector-java-5.1.35-bin.jar 分别复制到 hive/lib 与 Spark_CLASSPATH 目录下。

（2）用 hive/lib/jline -2.12.jar 替换 share/hadoop/yarn/lib7jline -0.9.94.jar，如果没有替换，运行./bin/hive 命令时会报错。

（3）在 hive-env.sh 中按实际集群情况配置下面三个变量：

```
export HADOOP_HOME="/home/dong/hadoop"
export HIVE_HOME="/home/dong/hive-0.13"
export HIVE_CONF_DIR = "/home/dong/hive-0.13/conf"
```

（4）配置 hive-site.xml。

```
<? xml-stylesheet type = "text/xsl" href=" configuration.xsl" ?>
<configuration>
<property>
<name>hive.metastore.local< /name >
<value>true</value>
</property>
<property>
<name>javax.jdo.option.ConnectionURL</name>
<value>jdbc:mysql://192.168.149.132:3306/hive?=createDatabaseIfNotExist=
true</value>
</property>
<property>
<name>javax.jdo.option.ConnectionDriverName</name>
<value>com.mysql.jdbc.Driver</value>
</property>
<property>
<name>javax.jdo.option.ConnectionUserName</name>
<value>dong</value>
</property>
<property>
<name>javax.jdo.option.ConnectionPassword</name>
<value>dong</value>
</property>
</coniiguration>
```

其中，javax.jdo.option. ConnectionURL 为数据库连接的 URL 信息，根据 MySQL 具体安装设置。示例中连接的用户名和密码为 dong/dong。

3. 启动 Hadoop

接下来启动 Hadoop。

4. 启动 Spark-shell

最后启动 Spark-shell。

6.6.2 构建数据仓库

学生信息管理系统包含了学生基本信息、院系信息、学生创新项目信息、学生学分成绩信息和奖学金信息。

1. 学生基本信息

包含学生姓名、学生 id、学生性别、学生年龄、入学年份、所在社团、所在院系 id 信息。

```
160201,Michael,male,37,2012,2,2
160101,Rose,female,33,2011,1,1
160301,justin,male,23,2013,3,3
160202,John,male,22,2012,2,2
160102,Lucy,female,27,2011,2,1
160203,Brewster,male,37,2012,1,2
160302,Susan,female,30,2013,1,3
160303,Justin,male,23,2013,3,3
160103,John,male,22,2011,2,1
160304,Lucy,female,27,2013,3,3
```

2. 院系信息

包含院系名称、编号信息。

```
Computer,1
Math,2
Art,3
```

3. 学生创新项目信息

包含学生 id、项目编号、起始年份、理论总学时、实践总学时、项目等级。

```
160201,XC2014001,2014,16,64,2
160101,XC201213,2012,32,48,3
160301,RW201103,2011,32,48,3
160202,XC2014002,2014,16,64,1
160102,XC2013002,2013,16,64,2
160102,XC2012011,2012,16,48,2
160302,RW201401,2014,32,32,2
160304,RW201503,2015,16,32,1
```

4. 学生学分成绩信息

包含学生 id、学分成绩信息。

```
160201,90
160101,83
160301,80
160202,70
160102,67
160102,89
160302,91
160303,58
160103,64
```

```
160304,77
```

5. 奖学金信息

包含学生 id、获奖年份、金额。

```
160201,2013,2000
160201,2014,3000
160102,2013,2000
160101,2013,2000
160301,2014,2000
160302,2014,2000
160302,2015,3000
160302,2016,4000
```

将学生信息管理系统的数据加载到 Hive 的 STUDENT 数据库中，并对学生信息管理系统的数据分别建表。

第 1 步，启动 Hadoop。

第 2 步，启动 Hive。

第 3 步，启动 Spark，进入 Spark-shell 执行后续步骤中的代码。

第 4 步，构建、使用 STUDENT 数据库。

1）使用 CREATE DATABASE 语句创建名为 STUDENT 的数据库，存放学生信息管理系统的数据。

```
scala>sqlContext.sql("CREATE DATABASE STUDENT")
```

2）使用学生信息管理系统的数据库 STUDENT。

```
scala>sqlContext.sql("USE STUDENT")
```

第 5 步，创建数据表。

仅在表不存在时创建（表存在时不创建）。

1）创建学生基础信息表 students。

```
scala>sqlContext.sql("CREATE TABLE IF NOT EXISTS students(id STRING, name STRING,gender STRING, age INT, year STRING, dep_id STRING) ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' LINES TERMINATED BY '\n'")
```

2）创建院系基础信息表 departments。

```
scala>sqlContext.sql("CREATE TABLE IF NOT EXISTS departments(name STRING, dep_id STRING)ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' LINES TERMINATED BY '\n'")
```

3）创建学生创新项目信息表 projects。

```
scala>sqlContext.sql("CREATE TABLE IF NOT EXISTS projects(id STRING, pro_id
```

```
STRING, year STRING, theory INT, practice INT, level INT) ROW FORMAT DELIMITED
FIELDS TERMINATED BY ',' LINES TERMINATED  BY '\n'")
```

4）创建学生学分成绩信息表 scores。

```
scala> sqlContext.sql("CREATE TABLE IF NOT EXISTS scores (id STRING, credit
INT) ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' LINES TERMINATED  BY '\n'")
```

5）创建学生奖学金信息表 scholarship。

```
scala>sqlContext.sql("CREATE TABLE IF NOT EXISTS scholarship(id STRING, year
STRING, amount INT) ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' LINES TERMINATED
BY '\n'")
```

6.6.3 加载数据

分别将本地文本数据加载到四个表中。

1. 加载学生基础信息表

```
scala>sqlContext.sql("LOAD DATA LOCAL INPATH '/home/dong/sqldata/students.
txt'OVERWRITE INTO TABLE students")
```

其中 OVERWRITE 表示覆盖当前表的数据，即先清除表数据，再将数据 insert 到表中。其他表的加载操作类似。

2. 加载院系基础信息表

```
scala>sqlContext.sql("LOAD DATA LOCAL INPATH '/home/dong/sqldata/
departments.txt' OVERWRITE INTO TABLE departments")
```

3. 加载学生创新项目信息表

```
scala>sqlContext.sql("LOAD DATA LOCAL INPATH '/home/dong/sqldata/projects.
txt' OVERWRITE INTO TABLE projects")
```

4. 加载学生学分成绩信息表

```
scala> sqlContext.sql("LOAD DATA LOCAL INPATH '/home/dong/sqldata/scores.
txt' OVERWRITE INTO TABLE scores")
```

5. 加载学生奖学金信息表

```
scala>sqlContext.sql("LOAD  DATA  LOCAL  INPATH  '/home/dong/sqldata/
scholarship.txt' OVERWRITE INTO TABLE scholarship")
```

6.6.4 查询数据

本节首先查看 STUDENT 库中各表的模式信息，然后针对学生信息管理系统中常见的查询功能在 Spark-shell 下给出具体实现。

1. 查看表的模式信息

查看 students 表。

```
scala>sqlContext.sql("select * from students")
```

运行结果：

```
res14: org.apache.spark.sql.DataFrame = [id: string, name: string, gender: string, age: int, year: string, dep_id: string]
```

查看 departments 表。

```
scala>sqlContext.sql("select * fromdepartments")
```

运行结果：

```
res15: org.apache.spark.sql.DataFrame = [name: string, dep_id: string]
```

查看 projects 表。

```
scala>sqlContext.sql("select * from projects")
```

运行结果：

```
res16: org.apache.spark.sql.DataFrame = [id: string, pro_id: string, year: string, theory: int, practice: int, level: int]
```

查看 scores 表。

```
scala> sqlContext.sql("select * from scores")
```

运行结果：

```
res17: org.apache.spark.sql.DataFrame = [id: string, credit: int]
```

查看 scholarship 表。

```
scala>sqlContext.sql("select * from scholarship")
```

运行结果：

```
res18: org.apache.spark.sql.DataFrame = [id: string, year: string, amount: int]
```

2. 统计各院系学生总数

首先根据院系 id 将表 students 与表 departments 进行 join 操作；然后根据 departments 中的院系名称进行分组；分组后针对 students 中唯一标识每个学生的 id 字段进行个数统计，最后得到各院系对应的学生总数。实现代码如下：

```
scala>sqlContext.sql("select b.name, count(a.id) from students a join departments b on a.dep_id=b.dep_id group by b.name").show
```

运行结果如图 6.29 所示。

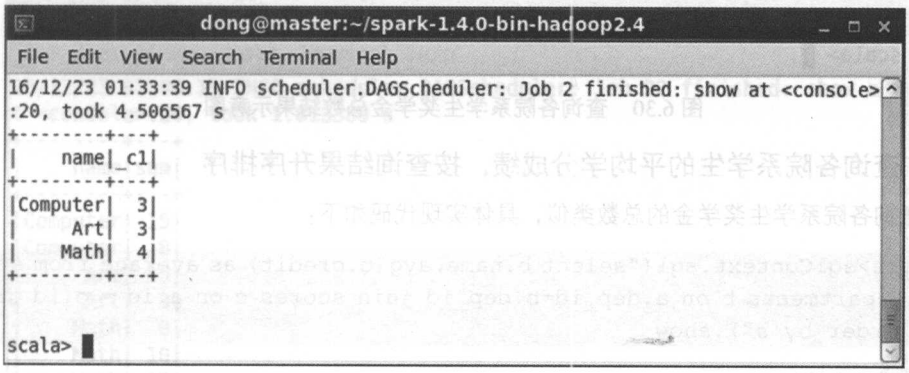


图 6.29 统计各院系学生总数结果示意图

3. 查询各院系学生奖学金的总数，按查询结果升序排序

首先根据院系 id 将表 students 与表 departments 进行 join 操作；然后将连接得到的表根据学生 id 再与表 salary 进行 join 操作；再根据表 departments 中的院系名进行分组；分组后针对学生的奖学金进行求和，默认即为升序排序。实现代码如下：

```
scala>sqlContext.sql("select b.name,sum(c.amount) as sum from students a join departments b on a.dep_id=b.dep_id join scholarship c on a.id = c.id group by b.name order by sum").show
```

运行结果如图 6.30 所示。

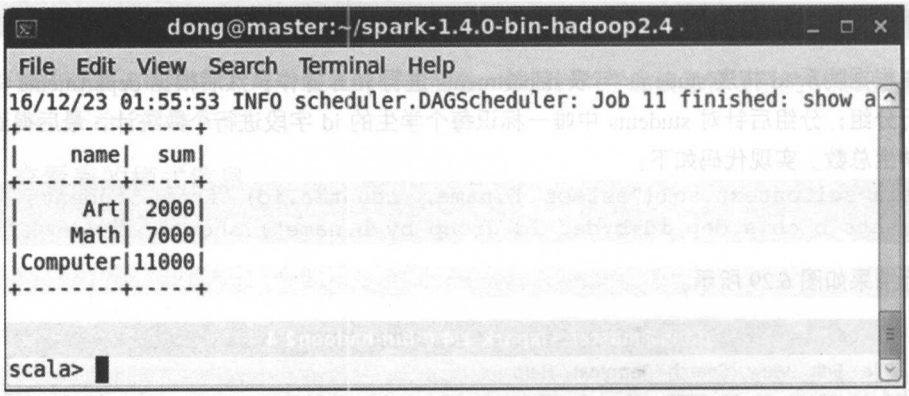


图 6.30　查询各院系学生奖学金总数结果示意图

4．查询各院系学生的平均学分成绩，按查询结果升序排序

与查询各院系学生奖学金的总数类似，具体实现代码如下：

```
scala>sqlContext.sql("select b.name,avg(c.credit) as average from students
a join departments b on a.dep_id=b.dep_id join scores c on a.id = c.id group by
b.name order by s").show
```

运行结果如图 6.31。

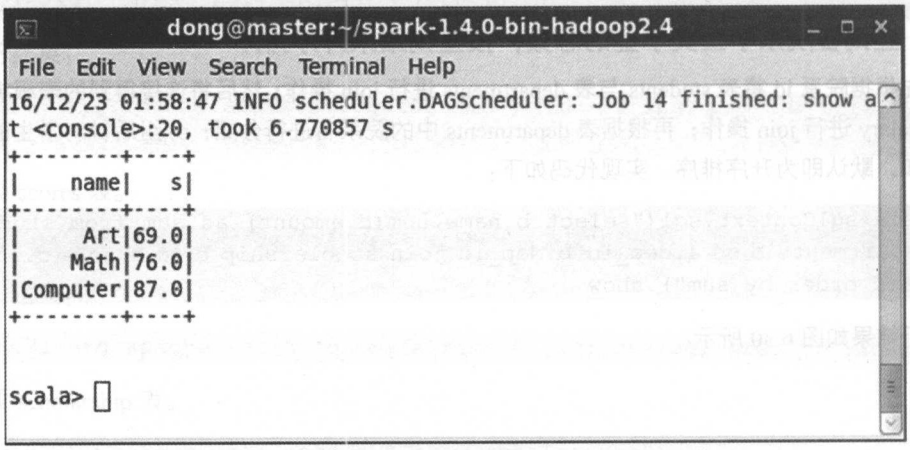


图 6.31　查询各院系学生的平均学分成绩结果示意图

5．统计各学院每年学生参与创新项目所获得的创新学分总数

首先根据学生 id 将表 projects 与表 students 进行 join 操作，进而得到表 h，其中包含了每个创新项目所对应的创新学分数，计算公式如下：

创新学分=项目等级*(项目理论课时+项目实践课时)/16

然后根据院系 id 将表 h 与表 departments 进行 join 操作；再结合院系名称与表 h 中的项目起始年份进行分组；最后对各组内数据进行累加求和。

```
scala>sqlContext.sql("select b.name,sum(h.pro_score) as sum, h.year from
(select a.id,p.year,p.level*(p.theory+p.practice)/16 as pro_score,a.dep_id
from projectsp join students a on a.id=p.id) h join departments b on
h.dep_id=b.dep_id group by b.name, h.year").show
```

运行结果如图 6.32 所示。

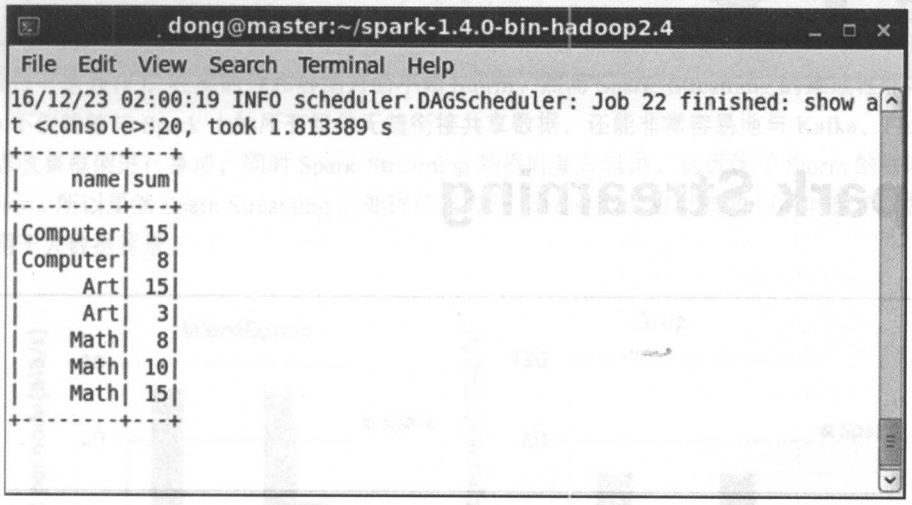


图 6.32 统计各学院每年学生参与创新项目所获得的创新学分总数结果示意图

第7章

Spark Streaming

7.1 Spark Streaming 概述

随着信息技术的迅猛发展，一方面数据量呈现出爆炸式增长趋势，另一方面数据的种类与变化速度也远远超出人们的想象，因此人们对大数据处理提出了更高的要求，越来越多的领域迫切需要大数据技术来解决领域内的关键问题。在一些特定的领域中（例如金融、灾害预警等），时间就是金钱、时间可能就是生命！然而传统的批处理框架却一直难以满足这些领域中的实时性需求。为此，涌现出了一批如 S4、Storm 的流式计算框架。Spark 是基于内存的大数据综合处理引擎，具有优秀的作业调度机制和快速的分布式计算能力，使其能够更加高效地进行迭代计算，因此 Spark 能够在一定程度上实现大数据的流式处理。

Spark Streaming 是 Spark 上的一个流式处理框架，可以面向海量数据实现高吞吐量、高容错的实时计算。Spark Streaming 支持多种类型数据源，包括 Kafka、Flume、trwitter、zeroMQ、Kinesis 以及 TCP sockets 等，如图 7.1 所示。Spark Streaming 实时接收数据流，并按照一定的时间间隔将连续的数据流拆分成一批批离散的数据集；然后应用诸如 map、reducluce、join 和 window 等丰富的 API 进行复杂的数据处理；最后提交给 Spark 引擎进行运算，得到批量结果数据，因此其也被称为准实时处理系统。



图 7.1 Spark Streaming 支持多种类型数据源

目前应用最广泛的大数据流式处理框架是 Storm。Spark Streaming 最低 0.5~2s 做一次处理（而 Storm 最快可达 0.1s），在实时性和容错方面不如 Storm。然而 Spark Streaming 的集成性非常好，通过 RDD 不仅能够与 Spark 上的所有组件无缝衔接共享数据，还能非常容易地与 Kafka、Flume 等分布式日志收集框架进行集成；同时 Spark Streaming 的吞吐量非常高，远远优于 Storm 的吞吐量，如图 7.2 所示。所以虽然 Spark Streaming 的处理延迟高于 Storm，但是在集成性与吞吐量方面的优势使其更适用于大数据背景。

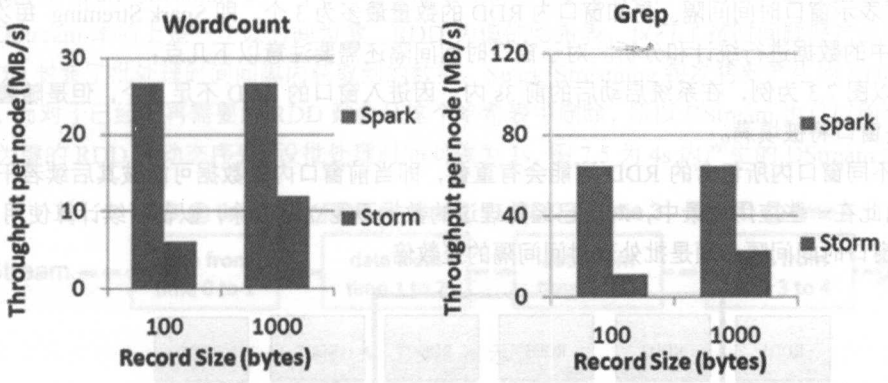


图 7.2 Spark Streaming 与 Storm 吞吐量比较图

7.2 Spark Streaming 基础概念

7.2.1 批处理时间间隔

在 Spark Streaming 中，对数据的采集是实时、逐条进行的，但是对数据的处理却是分批进行的。因此，Spark Streaming 需要设定一个时间间隔，将该时间间隔内采集到的数据统一进行处理，这个间隔称为批处理时间间隔。

也就是说对于源源不断的数据，Spark Streaming 是通过切分的方式，先将连续的数据流进行离

散化处理。数据流每被切分一次，对应生成一个 RDD，每个 RDD 都包含了一个时间间隔内所获取到的所有数据，因此数据流被转换为由若干个 RDD 构成的有序集合，而批处理时间间隔决定了 Spark Streaming 需要多久对数据流切分一次。Spark Streaming 是 Spark 上的组件，其获取的数据和数据上的操作最终仍以 Spark 作业的形式在底层的 Spark 内核中进行计算，因此批处理时间间隔不仅影响数据处理的吞吐量，同时也决定了 Spark Streaming 向 Spark 提交作业的频率和数据处理的延迟。需要注意的是，批处理时间间隔的设置将会伴随 Spark Streaming 应用程序的整个生命周期，无法在程序运行期间进行动态修改，所以需要综合考虑实际应用场景中的数据流特点和集群的处理性能等多种因素进行设定。

7.2.2 窗口时间间隔

窗口时间间隔又称为窗口长度，它是一个抽象的时间概念，决定了 Spark Streaming 对 RDD 序列进行处理的范围与粒度，即用户可以通过设置窗口长度来对一定时间范围内的数据进行统计和分析。如果设批处理时间设为 1s，窗口时间间隔为 3s，如 7.3 图所示，其中每个实心矩形表示 Spark Streaming 每 1 秒钟切分出的一个 RDD，若干个实心矩形块表示一个以时间为序的 RDD 序列，而透明矩形框表示窗口时间间隔。易知窗口内 RDD 的数量最多为 3 个，即 Spark Streming 每次最多对 3 个 RDD 中的数据进行统计和分析。对于窗口时间间隔还需要注意以下几点：

1. 以图 7.3 为例，在系统启动后的前 3s 内，因进入窗口的 RDD 不足 3 个，但是随着时间的推移，最终窗口将被填满。

2. 不同窗口内所包含的 RDD 可能会有重叠，即当前窗口内的数据可能被其后续若干个窗口所包含，因此在一些应用场景中，对于已经处理过的数据不能立即删除，以备后续计算使用。

3. 窗口时间间隔必须是批处理时间间隔的整数倍。

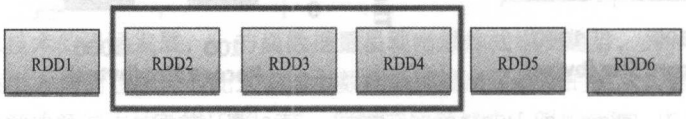


图 7.3　窗口时间间隔示意图

7.2.3 滑动时间间隔

滑动时间间隔决定了 Spark Streaming 对数据进行统计与分析的频率，多出现在与窗口相关的操作中。滑动时间间隔是基于批处理时间间隔提出的，其必须是批处理时间间隔的整数倍。在默认的情况下滑动时间间隔设置为与批处理时间间隔相同的值。如果批处理时间间隔为 1s，窗口间隔为 3s，滑动时间间隔为 2s，如图 7.4 所示，其含义是每隔 2s 对过去 3s 内产生的 3 个 RDD 进行统计分析。

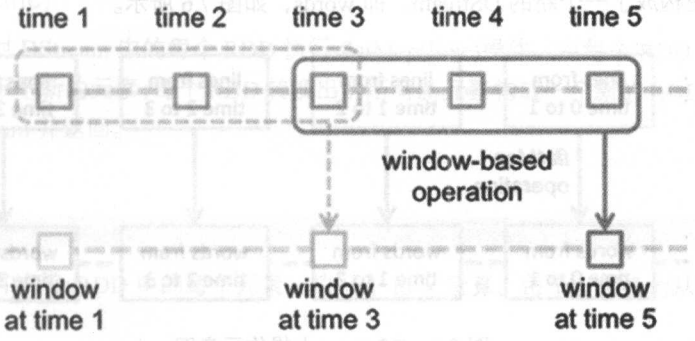


图 7.4 滑动时间间隔、窗口时间间隔、批处理时间间隔综合示意图

7.3 DStream 基本概念

DStream 是 Spark Streaming 的一个基本抽象，它以离散化的 RDD 序列的形式近似描述了连续的数据流。DStream 本质上是一个以时间为键，RDD 为值的哈希表，保存了按时间顺序产生的 RDD，而每个 RDD 封装了批处理时间间隔内获取到的数据。Spark Streaming 每次将新产生的 RDD 添加到哈希表中，而对于已经不再需要的 RDD 则会从这个哈希表中删除，所以 DStream 也可以简单地理解为以时间为键的 RDD 的动态序列。设批处理时间间隔为 1s，图 7.5 为 4s 内产生的 DStream 示意图。

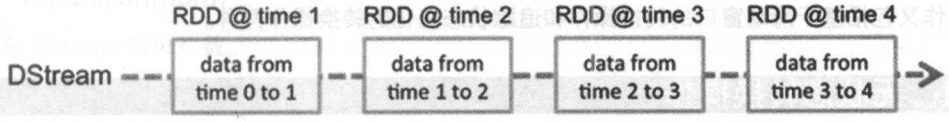


图 7.5 DStream 示意图

Spark Streaming 针对 DStream 提供了丰富的操作。需要注意的是，DStream 上的任何操作都会映射到其内部的每个 RDD 上，最终交由底层的 Spark 去执行。在示例 7-1 中，Spark Streaming 获取网络中指定主机上指定端口传输的数据，批处理时间间隔为 1s，在 4s 内一共切分出 4 个 RDD，这 4 个 RDD 对应的 DStream 记为 lines，其中每个 RDD 中包含若干个元素，均为 1s 内采集到的文本数据。

示例 7-1

```
val ssc = new StreamingContext(sparkConf, Seconds(1));
val lines = ssc.socketTextStream(serverIP, serverPort);
val words = lines.flatMap(_.split(""))
```

上述代码表示对 lines 执行 flatMap 操作。Spark Streaming 在执行该语句时将 flatMap 操作分别作用于 DStream 中的每个 RDD 上，当每个 RDD 执行完 flatMap(_.split(""))后，对应得到一个新的 RDD，

这些新的 RDD 便构成了一个新的 DStream，即 words，如图 7.6 所示。

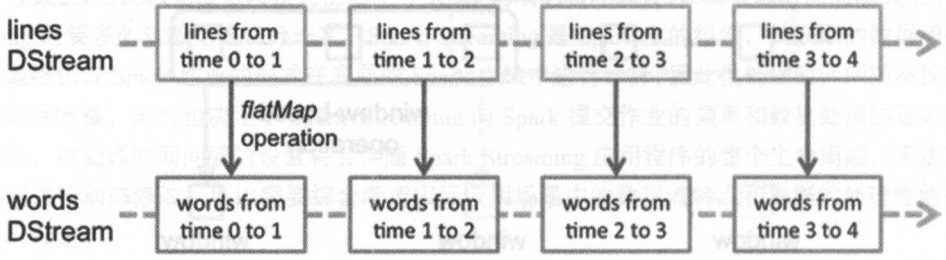


图 7.6　DStream 上操作示意图

DStream 上的操作提升了数据处理的抽象程度，隐藏了具体的实现细节，使得用户能够专注在 DStream 的业务逻辑上而无需关心其内部实现的细节。

7.4 DStream 的基本操作

DStream 的转换操作能够将一个 DStream 变换成另一个 DStream，具体又可分为无状态（stateless）和有状态（stateful）两种。

1. 在无状态转换操作中，对当前批次 RDD 的处理不依赖于之前批次的数据或中间结果。

2. 在有状态转换操作中，对当前批次 RDD 的处理依赖于之前批次的数据或中间结果。有状态转换操作又包括基于滑动窗口的转换操作和追踪状态变化的转换操作两种。

7.4.1 无状态转换操作

下面仅给出 DStream 部分常见的转换操作，这些操作与 RDD 中相应的转换操作功能类似，只是面向的粒度层面更高，但其本质仍然是通过对 RDD 中的每个元素进行运算实现相应的功能。

下面示例中令 ds 为创建的 DStream 实例。

1. map(func)

对 DStream 中的每个 RDD 执行 map(func)操作，对每个 RDD 中的每个元素进行 func 运算得到一个新的元素；每个 RDD 转换出的所有新元素构成一个新的 RDD；所有新的 RDD 构成一个新的 DStream 并返回。

示例 7-21

```
ds.map(x=>x+1)
```

上述代码将 ds 中每个 RDD 中的每个元素 x 执行 x+1 运算。

2. flatMap(func)

与 map 类似，对 DStream 中的每个 RDD 执行 flatMap(func)操作，对每个 RDD 中的每个元素进行 func 运算得到若干个新的元素；每个 RDD 转换出的所有新元素构成一个新的 RDD；所有新的 RDD 构成一个新的 DStream 并返回。

示例 7-3

```
ds.flatMap(x=>1 to x)
```

上述代码将 ds 中每个 RDD 中的每个元素 x 执行 1 to x 运算，每个 x 转换为从 1 到 x 的整数集合。

3. filter(func)

对 DStream 中的每个 RDD 执行 filter(func)操作，对每个 RDD 中的每个元素进行 func 运算，若计算结果为 true 值，则将该元素筛选出来；每个 RDD 筛选出的所有元素构成一个新的 RDD；所有新的 RDD 构成一个新的 DStream 并返回。

示例 7-4

```
ds.filter(x=>x==0)
```

上述代码对 ds 中每个 RDD 中的每个元素 x 判断是否为 0，若为 0 则将 x 作为新的 RDD 中的元素，否则将 x 过滤掉；所有新的 RDD 构成新的 DStream 并返回。

4. repartition(num)

改变 DStream 的分片数。

示例 7-5

```
ds.repartition(10)
```

上述代码将 ds 中每个 RDD 的分区数改为 10。

5. count()

对 DStream 中的每个 RDD 执行 count 操作并返回一个整数，即 RDD 中元素的个数；每个整数被封装为一个新的 RDD；所有新的 RDD 构成一个新 DStream 并返回。

示例 7-6

```
ds.count()
```

上述代码为查看 ds 中每个 RDD 中元素的个数。

6. reduce(func)

对 DStream 中每个 RDD 执行 reduce(func)操作，对每个 RDD 中的元素进行 func 聚合运算；每

个 RDD 计算出的结果构成一个新的 RDD；所有新的 RDD 构成一个新的 DStream 并返回。

示例 7-7

```
ds.reduce((x,y)=>x+y)
```

上述代码将 ds 中每个 RDD 中的所有元素执行累加操作。

7．reduceByKey(func)

针对 Key-Value 形式的 DStream，对每个 RDD[(K, V)]执行 reduceByKey(func)操作，对每个 RDD[(K, V)]中各组 Value 进行 func 聚合运算；每个 RDD[(K, V)]计算出的结果构成一个新的 RDD[(K, V)]；所有的新的 RDD[(K, V)] 构成一个新的 DStream 并返回。

示例 7-8

```
ds.reduceByKey((x,y)=>x+y)
```

上述代码将 ds 中每个 RDD 中的所有元素按照 Key 值进行分组，并对组内的 Value 值执行累加操作。

8．transform(func)

该操作是 DStream 提供的一个高级转换操作，其处理数据的抽象程度更高，允许用户为 DStream 定义一个 RDD 到 RDD 的函数 func。对 DStream 中的每个 RDD 执行 func 运算生成一个新的 RDD，所有新的 RDD 构成一个新的 DStream。transform()的一个常见应用是重用为 RDD 编写的批处理代码。

7.4.2 有状态转换操作

1．window(windowLength, slideInterval)

将原 DStream 按所指定的窗口进行切分返回新的 DStream。每间隔 slideInterval 时间单位，将 DStream 中前 windowLength 时间单位内的数据封装成新的 RDD；所有新的 RDD 构成一个新的 DStream 并返回。

示例 7-9

```
ds.window(Seconds(30), Seconds(20))
```

上述代码每隔 20s 将 ds 中前 30s 内的 RDD 封装为一个新的 RDD，所有新的 RDD 构成新的 DStream。

2．reduceByWindow(func, windowLength, slideInterval)

每间隔 slideInterval 时间单位，将 DStream 中前 windowLength 时间单位内所有 RDD 中的元素进行 func 聚合运算，计算出的结果构成一个新的 RDD；所有新的 RDD 构成一个新的 DStream。

示例 7-10

```
ds.reduceByWindow(x=>x!=0, Seconds(30), Seconds(20))
```

上述代码每隔 20s 将 ds 中前 30s 内的 RDD 封装为一个新 RDD，所有的新 RDD 构成新 DStream。

3. reduceByKeyAndWidow(func, windowLength, slideInterval)

该操作针对 Key-Value 形式的 DStream[(K, V)]，功能与 reduceByWindow 类似。每间隔 slideInterval 时间单位，将前 windowLength 时间单位内所有 RDD[(K, V)]中的 Value 按 Key 分组执行 func 运算，并将结果封装成一个新的 RDD；所有新的 RDD 构成一个新的 DStream[(K, V)]并返回。

4. updateStateByKey(func)

该操作针对 Key-Value 形式的 DStream[(K, V)]，其中 func 是状态更新函数，指定了如何根据新的事件更新每个键对应状态，通过 func 构建出一个新的 DStream[(K,V)]并返回。

使用 updateStateByKey 操作通常需要两个步骤：

1）定义状态：可以是任意数据类型。

2）定义状态更新函数：指定如何使用先前的状态和当前批次的数据计算出新的状态值。

5. countByWindow

基于窗口统计 DStream 中满足条件的元素数量，功能与 count 类似。每间隔 slideInterval 时间单位，统计前 windowLength 时间单位内所有 RDD 中元素的个数，并将统计结果封装成一个新的 RDD；所有新的 RDD 构成一个新的 DStream 并返回。

7.4.3 输出操作

1. print()

将 DStream 中的前 10 个元素打印出来。print 是一种常用的调试性输出操作。

2. foreachRDD(func)

对 DStream 中的每个 RDD 执行 func 函数运算，其中 func 是一个 RDD 到 RDD 的映射函数，进而得到新的 DStream。该操作是 DStream 通用的输出操作。

示例 7-11

```
//利用 foreachRDD 将数据存储到外部系统
ds.foreachRDD { rdd =>
                    rdd.foreachPartition{
//打开到存储系统的连接（比如一个数据库的连接）
                        partition.foreach { item =>……
//使用连接把 item 存到系统中
                    }
//关闭连接
```

```
        }
    }
```

3. saveAsTextFiles(prefix,[suffixSx])

将 DStream 中的内容保存为文本文件，其中每次批处理时间间隔内产生的文件以 prefix-TIME_IN_MS[.suffix]命名。

示例 7-12

```
df. saveAsTextFiles("outputDir", "txt")
```

7.4.4 持久化操作

通过 DStream 显示调用函数 persist 可以将其中的 RDD 缓存在内存中。当 DStream 的内容需要被多次计算的时候，这种做法会极大地提高执行效率。对于一些窗口操作，例如 reduceByWindow、reduceByKeyAndWindow 以及基于状态的操作，例如 updateStateByKey 会隐式地将 DStream 中的 RDD 持久化在内存中。

与 Spark 中的 RDD 类似，Spark Streaming 对于 DStream 中的数据也有存储级别，当数据来自磁盘的时候，默认存储级别为 MEMORY_ONLY_SER，含义是“内存+序列化”。当数据来自网络时（Flume Kafka Socket），默认存储级别为 MEMORY_AND_DISK_SER_2，其含义是“内存+磁盘+序列化+备份 2 份”。由于网络中的数据流不能像磁盘上的数据一样可以重复加载，因此出于容错方面考虑，对于接收到的网络数据需要持久化到磁盘，否则出错时难以重构数据。因此对于窗口、状态相关的操作必须进行 checkpoint 处理，可以通过 StreamingContext 的 checkpoint 来指定持久化目录并制定时间间隔。

7.5 数据源

Spark Streaming 应用程序的处理流程是首先获取实时数据流，再将获取到的数据流转换为 DStream，之后所有的业务逻辑都是基于 DStream 展开。无论 Spark Streaming 应用程序多么复杂，均以数据输入源作为起点。本节将介绍 Spark Streaming 能够获取哪些类型的数据。

7.5.1 基础数据源

所有用来从核心数据源创建 DStream 的方法都封装在 Streaming Context 中，其中主要包含套接字、文件系统与 Akka actor。

1. 套接字

通过 Streaming Context 对象调用 socketTextStream()方法实现对指定主机名指定端口上数据流的

获取。

示例 7-13

```
val ds= ssc.socketTextStream("localhost", 7777)
```

其中 ssc 为 Streaming Context 对象，SparkStreaming 将从本地主机的 7777 端口获取数据创建 DStream。

2. 文件流

Spark 支持从任意 Hadoop 兼容的文件系统中读取数据创建 DStream。在实际应用场景中，文件流是Spark Streaming处理的一种主要数据源类型，尤其是对于日志这类需要存储到HDFS上的数据。SparkStreaming可以通过Streaming Context对象调用textTextStream()方法实现对指定目录下文本数据流的获取，该方法同样可以从其他文件系统(HDFS、S3、NFS 等)的文件中读取数据创建一个 DStream。

示例 7-14

```
val ds= ssc.textFileStream(dataDirectory)
```

其中 ssc 为 Streaming Context 对象，读取 dataDirectory 目录下的文件流创建 DStream。

需要注意，当 Spark Streaming 监听 dataDirectory 目录下的文件时，读取的必须是具有相同数据格式的文件；不支持在嵌套目录下读取文件；以新文件（新创建的文件、重命名的文件、新移入 dataDirectory 目录下的文件）为单位读取数据，无法获取已有文件中追加的数据。

3. Akka actor 流

Spark Streaming 也可以基于自定义 actors 的流创建 DStream，通过 Akka actors 接收数据流，可以使用方法 actorStream(actorProps, actorname)实现，由于篇幅限制，本书对其不做深入介绍。

7.5.2 高级数据源

Spark Streaming 可以基于多种数据源创建 DStream，但对其中一些数据源的处理需要借助 Spark 库以外的接口和添加一些相应的依赖，我们称这些数据源为高级数据源。高级数据源主要有以下几种：

1. Twitter：Spark Streaming 的 TwitterUtils 工具类使用 Twitter4J 库。
2. Flume：Spark Streaming 可以从 Flume 中接收数据。
3. Kafka：Spark Streaming 可以从 Kafka 中接收数据。
4. Kinesis：Spark Streaming 可以从 Kinesis 中接收数据。

需要注意的是，在 Spark Shell 中难以基于高级数据源直接创建 DStream。若在 Spark Shell 中对面向高级数据源的应用程序进行测试，需要下载相应的 Maven 工程的 Jar 依赖并添加到类路径中。

7.6 Spark Streaming 编程模式与案例分析

7.6.1 Spark Streaming 编程模式

下面以 Spark Streaming 官方提供的 WordCount 代码为例来介绍 Spark Streaming 的使用方式。

示例 7-15

```
import org.apache.spark._
import org.apache.spark.streaming._
import org.apache.spark.streaming.StreamingContext._
/*创建一个本地模式的 StreamingContext，并设定 master 节点工作线程数为 2，并以 1 秒作为批处理时间间隔。*/
val conf =  new SparkConf().setMaster("local[2]").
            setAppName("NetworkWordCount")
val ssc = new StreamingContext(conf, Seconds(1))
/*通过获取"localhost"节点 9999 端口中的实时数据流创建 DStream。*/
val lines = ssc.socketTextStream("localhost", 9999)
/*以空格作为分割 DStream 中数据的依据，使得每一行文本转换为若干个单词。*/
val words = lines.flatMap(_.split(" "))
import org.apache.spark.streaming.StreamingContext._
/*对于 words 中的每个单词 word，转换为相应的二元组形式(word,1)，在此基础上统计每个单词出现的次数。*/
val pairs = words.map(word => (word, 1))
val wordCounts = pairs.reduceByKey(_ + _)
//输出 DStream 中每个 RDD 中前 10 个元素。
wordCounts.print()
//启动 Spark Streaming 应用程序。
ssc.start()
//等待计算完成。
ssc.awaitTermination()
```

Spark Streaming 应用程序在功能结构上通常包含以下五部分，如上述示例 7-15 所示。

1. 导入 Spark Streaming 相关包

Spark Streaming 作为 Spark 框架上的一个组件，具有很好的集成性。在开发 Spark Streaming 应用程序时，只需导入 Spark Streaming 相关包，无需额外的参数配置。

2. 创建 StreamingContext 对象

同 Spark 应用程序中的 SparkContext 对象一样， StreamingContext 对象是 Spark Streaming 应用程序与集群进行交互的唯一通道，其中封装了 Spark 集群的环境信息和应用程序的一些属性信息。

在该对象中通常需要指明应用程序的运行模式（示例 7-15 中设为 local[2]）、设定应用程序名称（示例 7-15 中设为 NetworkWordCount）、设定批处理时间间隔（示例 7-15 中设为 Seconds(1)即 1 秒钟），其中批处理时间间隔需要根据用户的需求和集群的处理能力进行适当地设置。

3. 创建 InputDStream

Spark Streaming 需要根据数据源类型选择相应的创建 DStream 的方法。示例 7-15 中 Spark Streaming 通过 StreamingContext 对象调用 socketTextStream 方法处理以 socket 连接类型数据源，创建出 DStream 即 lines。Spark Streaming 同时支持多种不同的数据源类型，其中包括 Kafka、Flume、HDFS/S3、Kinesis 和 Twitter 等数据源。

4. 操作 DStream

对于从数据源得到的 DStream，用户可以调用丰富的操作对其进行处理。示例 7-15 中针对 lines 的一系列操作就是一个典型的 WordCount 执行流程：对于当前批处理时间间隔内的文本数据以空格进行切分，进而得到 words；再将 words 中每个单词转换为二元组，进而得到 pairs；最后利用 reduceByKey 方法进行统计。

5. 启动与停止 Spark Streaming 应用程序

在启动 Spark Streaming 应用程序之前，DStream 上所有的操作仅仅是定义了数据的处理流程，程序并没有真正连接上数据源，也没有对数据进行任何操作，当 ssc.start()启动后程序中定义的操作才会真正开始执行。

7.6.2 文本文件数据处理案例（一）

1. 功能需求

实时监听并获取本地 home/dong/Streamingtext 目录中新生成的文件（文件均为英文文本文件，单词之间使用空格进行间隔），并对文件中各单词出现的次数进行统计。

2. 代码实现

```
package dong.spark
import org.apache.spark.SparkConf
import org.apache.spark.streaming.{Seconds,StreamingContext}
import org.apache.spark.streaming.StreamingContext._
object StreamingFileWordCount {
  def main(args: Array[String]): Unit ={
//以 local 模式运行，并设定 master 节点工作线程数为 2。
    val sparkConf = new SparkConf().
                    setAppName("StreamingFileWordCount").
                    setMaster("local[2]")
/*创建 StreamingContext 实例，设定批处理时间间隔为 20 秒。*/
```

```
    val ssc = new StreamingContext(sparkConf,Seconds(20))
/*指定数据源来自本地 home/dong/Streamingtext。*/
    val lines = ssc.textFileStream("/home/dong/Streamingtext")
/*在每个批处理时间间隔内，对指定文件夹中变化的数据进行单词统计并打印。*/
    val words= lines.flatMap(_.split(" "))
    val wordcounts=words.map(x=>(x,1)).reduceByKey(_+_)
    wordcounts.print()
    ssc.start()
    ssc.awaitTermination()
  }
}
```

3. 运行演示

第 1 步，启动 Hadoop 与 Spark。

```
$ start-all.sh
$ cd spark-1.4.0-bin-hadoop2.4
$ sbin/start-all.sh
```

第 2 步，创建 Streaming 监控目录。

```
$ mkdir /home/dong/Streamingtext
```

在 dong 用户主目录下创建 Streamingtext 为 Spark Streaming 监控的目录，创建后如图 7.7 所示。

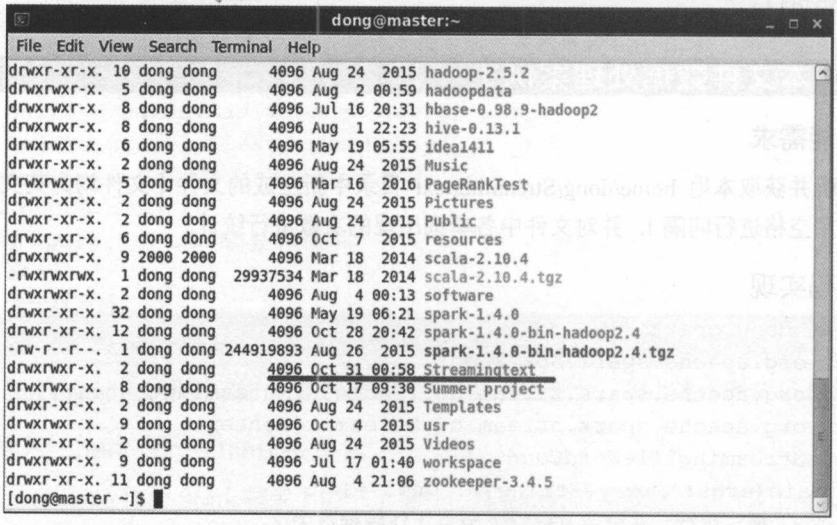


图 7.7　dong 用户主目录下创建 Streamingtext 文件夹

第 3 步，在 IntelliJ IDEA 中编辑运行 Streaming 程序。

在 IntelliJ IDEA 中创建工程 StreamingFileWordCount（IntelliJ IDEA 中创建工程详见 4.3 节），编

辑对象 StreamingFileWordCount，如图 7.8 所示。

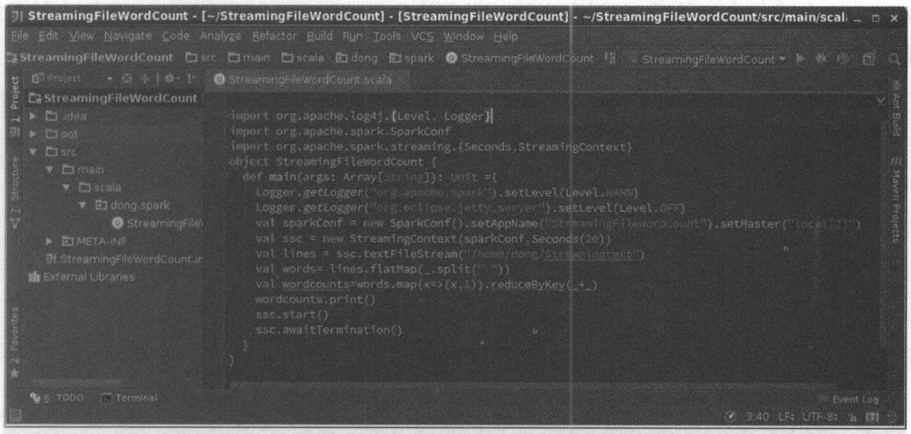


图 7.8 IntelliJ IDEA 中 StreamingFileWordCount 示意图

由于该示例没有输入参数，因此不需要配置参数，可直接单击右键->单击"Run'StreamingFileWordCount' "。

第 4 步，在监听目录下创建文本文件。

在 master 节点上的/home/dong/Streamingtext 中分别创建 file1.txt 与 file2.txt。

file1.txt 内容如下：

```
aa
bb
```

file2.txt 内容如下：

```
ee
dd
cc
```

创建后，/home/dong/Streamingtext 中内容如图 7.9 所示。

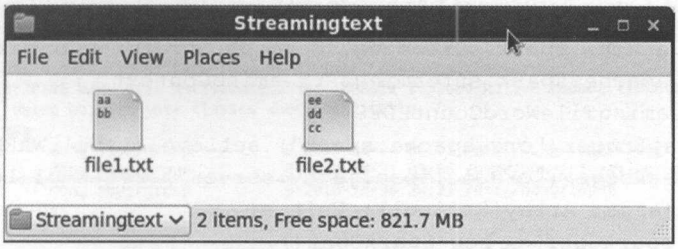


图 7.9 Streamingtext 文件夹内容示意图

4. 查看结果

终端窗口输出了每个批处理时间间隔（20 秒）内，/home/dong/Streamingtext 中新生成文件所包含的各单词个数，如图 7.10 所示。

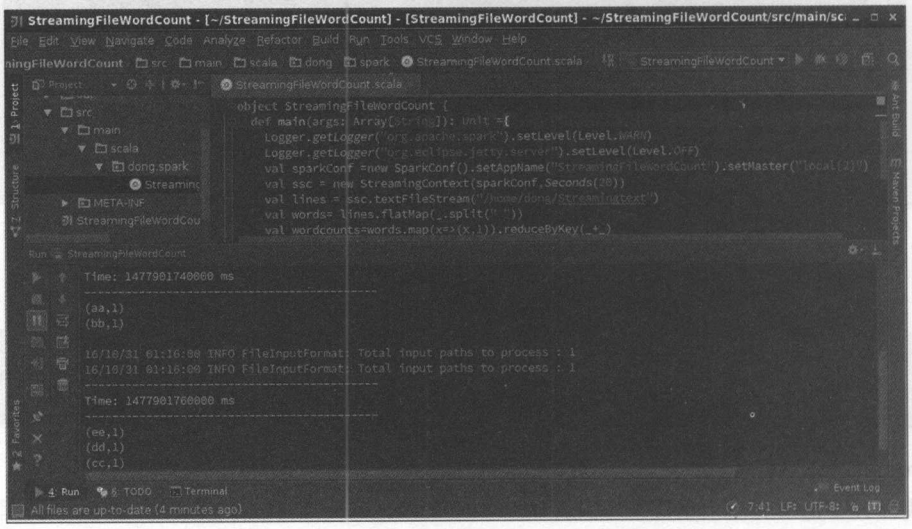


图 7.10　StreamingFileWordCount 运行结果示意图

7.6.3　文本文件数据处理案例（二）

1. 功能需求

实时监听并获取 HDFS 上 user/dong/Streamingtext 目录中生成的文件（文件均为英文文本文件，单词之间使用空格进行间隔），并对文件中各单词出现的次数进行统计。

2. 代码实现

```
package dong.spark
import org.apache.log4j.{Level, Logger}
import org.apache.spark.SparkConf
import org.apache.spark.streaming.{Seconds,StreamingContext}
import org.apache.spark.streaming.StreamingContext._
object StreamingFileWordCountHDFS {
  Logger.getLogger("org.apache.spark").setLevel(Level.WARN)
  Logger.getLogger("org.eclipse.jetty.server").setLevel(Level.OFF)
  def main(args: Array[String]): Unit ={
      val sparkConf =  new SparkConf().
                       setAppName("StreamingFileWordCountHDFS").
                       setMaster("spark://192.168.149.132:7077")
```

```
    //创建 StreamingContext 实例，设定批处理时间间隔为 20 秒。
       val ssc = new StreamingContext(sparkConf,Seconds(20))
    //指定数据源来自 HDFS 上 user/dong/Streamingtext。
       val                                    lines                                    =
ssc.textFileStream("hdfs://master:9000/user/dong/Streamingtext")
    //对指定文件夹中变化的数据进行单词统计并且打印。
       val words= lines.flatMap(_.split(" "))
       val wordcounts=words.map(x=>(x,1)).reduceByKey(_+_)
       wordcounts.print()
       ssc.start()
       ssc.awaitTermination()
     }
    }
```

3. 运行演示

第 1 步，启动 Hadoop 与 Spark

```
$ start-all.sh
$ cd spark-1.4.0-bin-hadoop2.4
$ sbin/start-all.sh
```

第 2 步，创建 Streaming 监听目录

```
$ hdfs dfs -mkdir /user/dong/Streamingtext
```

HDFS 上创建/user/dong/Streamingtext 为 Streaming 应用程序的监听目录，创建后如图 7.11 所示。

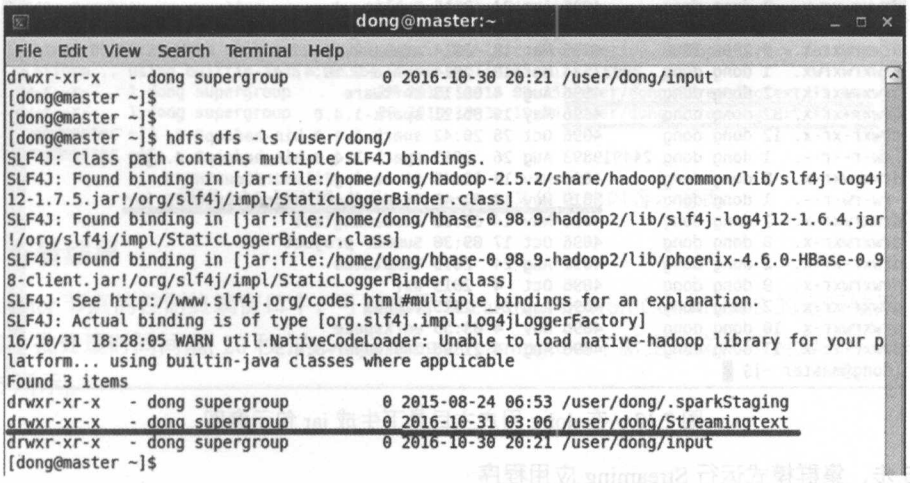


图 7.11 HDFS 上 Streamingtext 文件夹示意图

第 3 步，在 IntelliJ IDEA 中编辑 Streaming 程序

在 master 节点启动 IntelliJ IDEA，创建工程 StreamingFileWordCountHDFS，如图 7.12 所示。

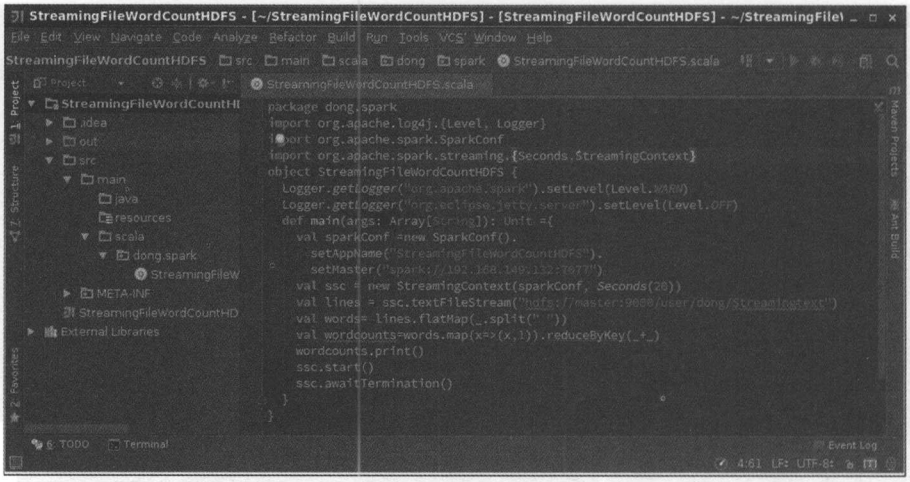


图 7.12　IntelliJ IDEA 中 StreamingFileWordCountHDFS 示意图

第 4 步，打包

将应用程序打包至 master 节点 dong 用户主目录下（打包过程详见 4.3 节），如图 7.13 所示。

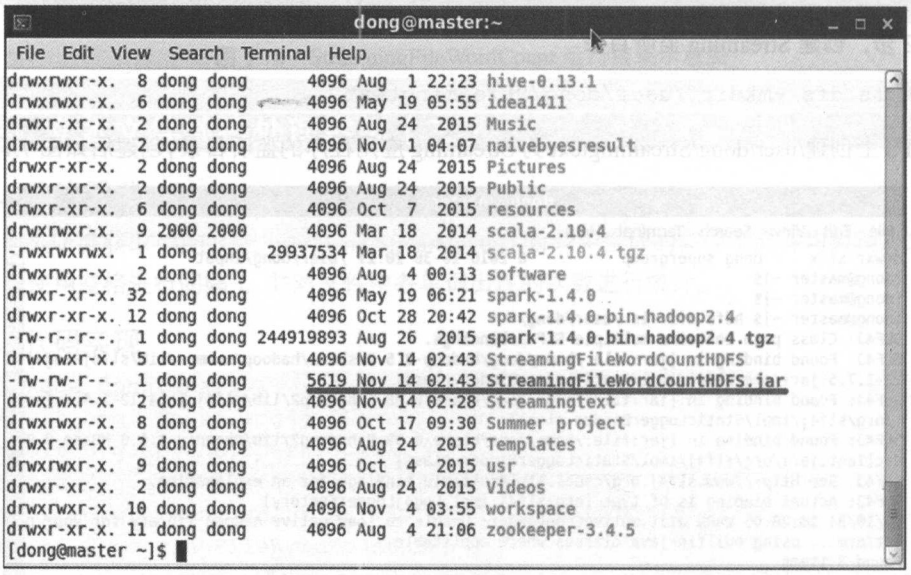


图 7.13　在 dong 用户主目录下生成 jar 包示意图

第 5 步，集群模式运行 Streaming 应用程序

在 master 节点开启终端，进入 spark-1.4.0-bin-hadoop2.4 文件夹，执行下面命令向 Spark 集群提交应用程序。

```
$ bin/spark-submit /home/dong/StreamingFileWordCountHDFS.jar
```

第 6 步，上传文件至 HDFS 上的监听目录中

master 节点 dong 用户主目录下创建文本文件 f1.txt 与 f2.txt。

f1.txt 内容如下：

```
this is f1
f1 spark
f1 world
```

f2.txt 内容如下：

```
this is f2
f2 spark
f2 world
```

分别将 f1.txt 和 f2.txt 上传至 HDFS 上/user/dong/Streamingtext 中。

```
$ hdfs dfs -put /home/dong/f1.txt /user/dong/Streamingtext
$ hdfs dfs -put /home/dong/f2.txt /user/dong/Streamingtext
```

上传后 HDFS 的/user/dong/Streamingtext 文件夹中的内容如图 7.14 所示。

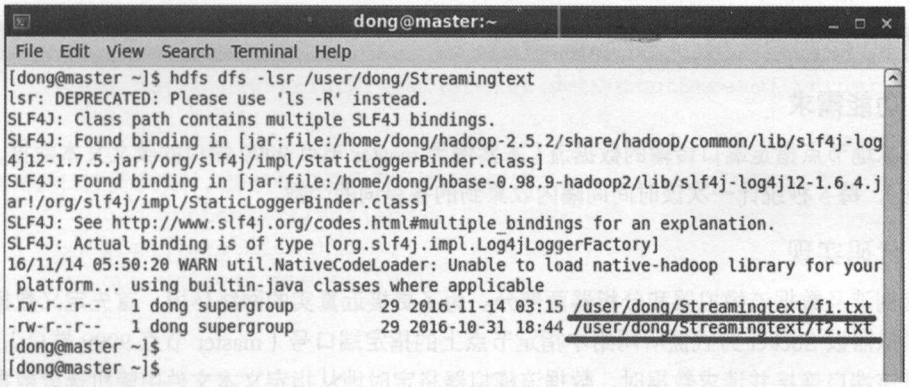


图 7.14 HDFS 上 Streamingtext 文件夹的内容示意图

4. 查看结果

在 master 节点的控制终端窗口中，可以查看 StreamingFileWordCountHDFS 的运行结果。窗口输出了在每个批处理时间间隔(20 秒)内，/user/dong/Streamingtext 中新生成文件所包含的各单词个数，如图 7.15 所示。

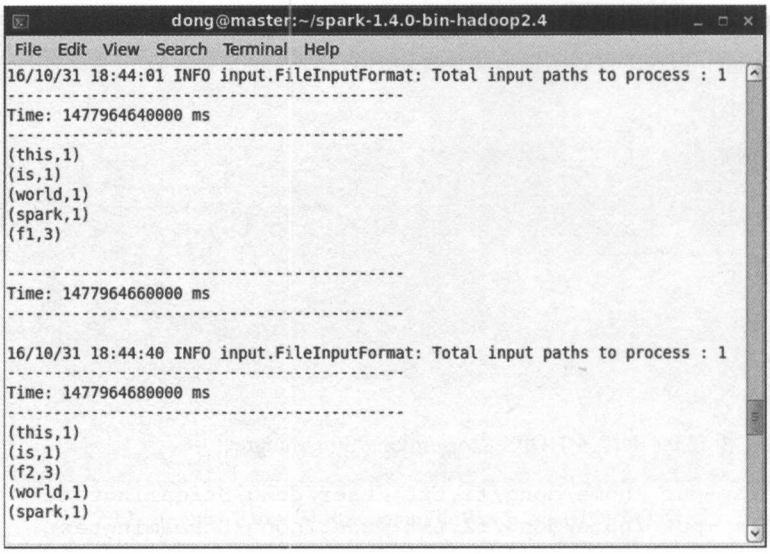


图 7.15　StreamingFileWordCountHDFS 运行结果示意图

7.6.4　网络数据处理案例（一）

1．功能需求

监听本地节点指定端口传输的数据流（本案例为 master 节点 9999 端口的英文文本数据，以逗号间隔单词），每 5 秒统计一次该时间间隔内收集到的各单词的个数。

2．代码实现

本案例涉及数据流模拟器和分析器两部分。为了更接近真实的网络环境，首先定义数据流模拟器，该模拟器以 Socket 方式监听网络中指定节点上的指定端口号（master 节点 9999 端口），当外部程序通过该端口连接并请求数据时，数据流模拟器将定时地从指定文本文件中随机选取数据发送至指定端口（每间隔 1 秒钟数据流模拟器从 master 节点上的/home/dong/Streamingtext/file1.txt 中随机截取一行文本发送给 master 节点的 9999 端口），通过这种方式模拟网络环境下源源不断的数据流。针对获取到的实时数据，再定义分析器（Spark Streaming 应用程序），用以统计时间间隔（5 秒）内收集到的单词个数。

数据流模拟器代码实现如下：

```
package dong.spark
import java.io.{PrintWriter}
import java.net.ServerSocket
import scala.io.Source
objectSocketSimulation {
//定义随机获取整数的方法。
  def index(length: Int)={
```

```
    import java.util.Random
    val rdm = new Random
    rdm.nextInt(length)
  }
  def main(args:Array[String]): Unit ={
    if(args.length!=3){
/*调用数据流模拟器需要三个参数：文件路径、端口号和批处理时间间隔时间（单位：毫秒）。*/
      System.err.println("Usage:<filename><port><millisecond>")
      System.exit(1)
    }
//获取指定文件总的行数。
    val filename = args(0)
    val lines = Source.fromFile(filename).getLines().toList
    val filerow=lines.length
//指定监听参数 args(1)指定的端口，当外部程序请求时建立连接。
    val listener =new ServerSocket(args(1).toInt)
    while(true){
      val socket = listener.accept()
      new Thread(){
        override def run={
          println("Got client connected from:"+socket.getInetAddress)
          val out = new PrintWriter(socket.getOutputStream(),true)
          while(true){
            Thread.sleep(args(2).toLong)
//当该端口接受请求时，随机获取某行数据发送给对方。
            val content= lines(index(filerow))
            println(content)
            out.write(content+'\n')
            out.flush()
          }
          socket.close()
        }
      }.start()
    }
  }
}
```

分析器代码如下：

```
package dong.spark
import org.apache.spark.streaming.{Milliseconds,Seconds, StreamingContext}
import org.apache.spark.streaming.StreamingContext._
import org.apache.spark.storage.StorageLevel
object NetworkWordCount {
  def main (args:Array[String]) ={
```

```
    //以 local 模式运行，并设定 master 节点工作线程数为 2。
      val conf=new SparkConf().setAppName("NetworkWordCount").
               setMaster("local[2]")
      val sc=new SparkContext(conf)
      val ssc=new StreamingContext(sc, Seconds(5))
    /*通过 socketTextStream 获取指定节点指定端口的数据创建 DStream,并保存在内存和硬盘中,
其中节点与端口分别对应参数 args(0)和 args(1)。*/

      val lines=ssc.socketTextStream(args(0),
                        args(1).toInt,
                        StorageLevel.MEMORY_AND_DISK_SER)
    //在每个批处理时间间隔内对获取到的数据进行单词统计并且打印。
      val words= lines.flatMap(_.split(","))
      val wordcounts = words.map(x=>(x,1)).reduceByKey(_+_)
      wordcounts.print()
      ssc.start()
      ssc.awaitTermination()
    }
  }
```

3. 运行演示

第 1 步，在 IntelliJ IDEA 中编辑运行 Streaming 程序。

master 节点启动 IntelliJ IDEA，创建工程 NetworkWordCount，编辑模拟器与分析器。模拟器如图 7.16 所示，分析器如图 7.17 所示。

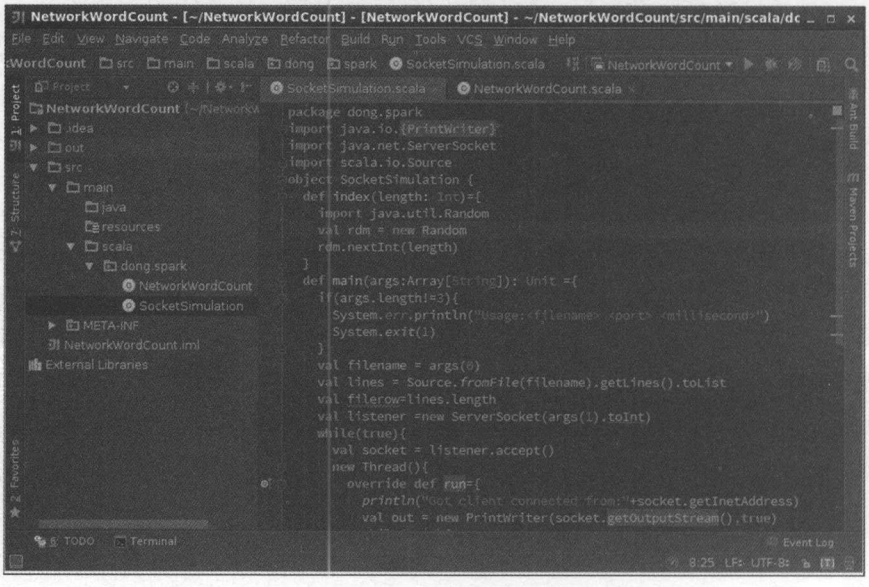


图 7.16　IntelliJ IDEA 中数据流模拟器示意图

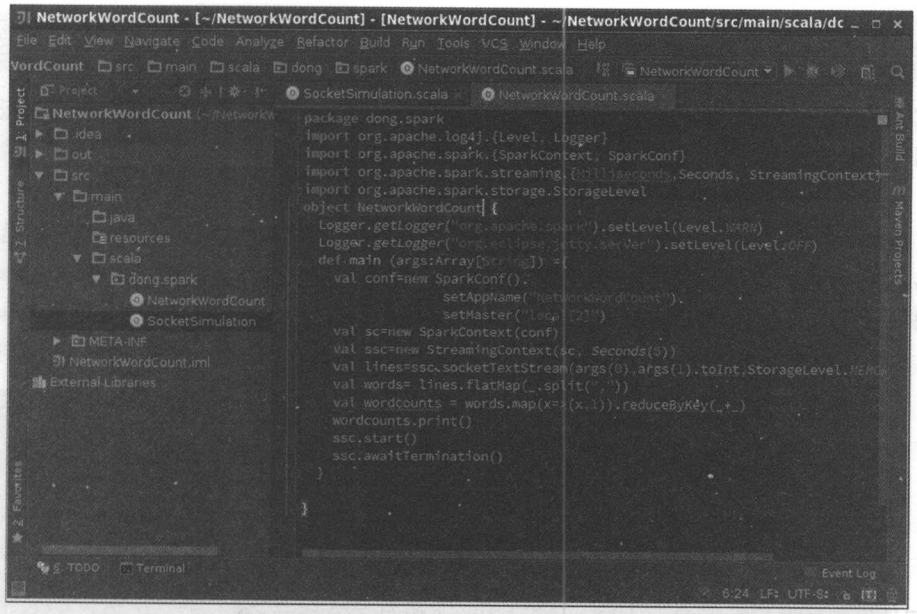


图 7.17 IntelliJ IDEA 中分析器示意图。

第 2 步，创建模拟器数据源文件。

在 master 节点创建/home/dong/Streamingtext 目录，在其中创建文本文件 file1.txt。

file1.txt 内容如下：

```
spark,
hello,
hbase,
world,
```

第 3 步，打包数据流模拟器。

打包过程详见本书 4.3.3 节。在 Artifacts 打包配置界面中，根据用户实际 scala 安装目录，在 Class Path 中添加下述 scala 依赖包，如图 7.18 所示。

```
/usr/scala-2.10.4/lib/scala-swing.jar
/usr/scala-2.10.4/lib/scala-library.jar
/usr/scala-2.10.4/lib/scala-actors.jar
```

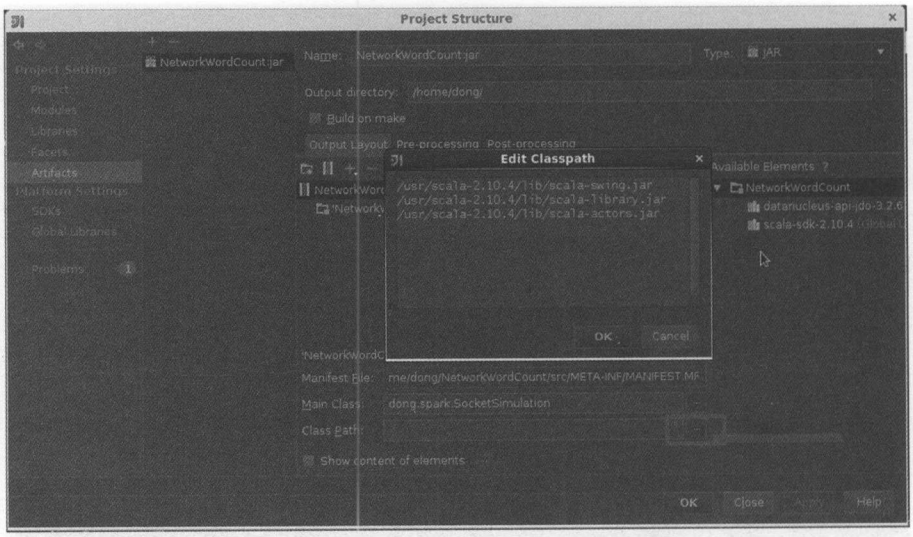


图 7.18　在 Class Path 中添加 scala 依赖包

打包后在主目录下生成 NetworkWordCount.jar，如图 7.19 所示。

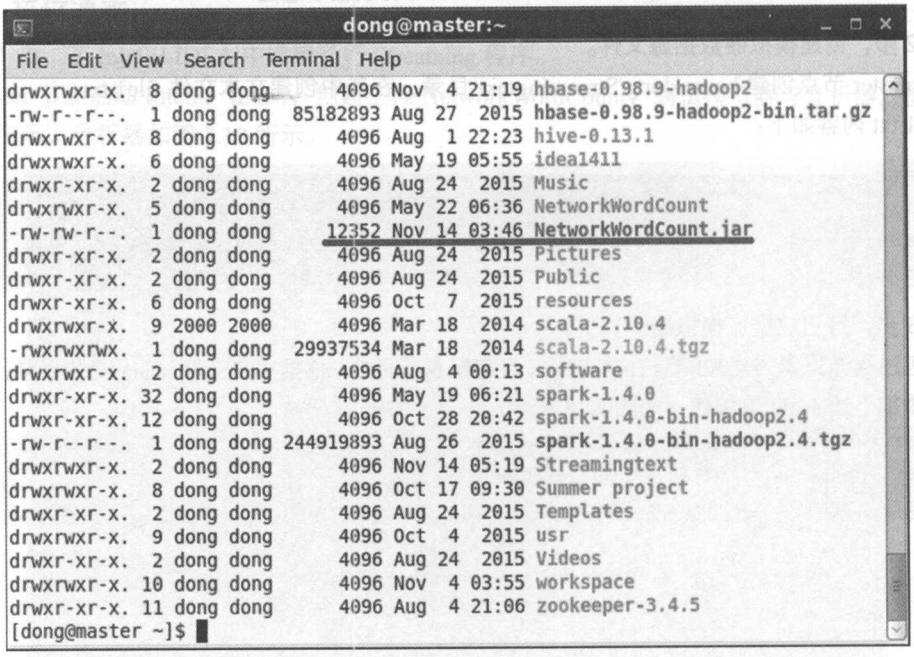


图 7.19　在 dong 用户主目录下生成 NetworkWordCount.jar 示意图

第 4 步，启动数据流模拟器。

在 master 节点开启控制终端，通过下面代码启动数据流模拟器。

```
    $ java -cp /home/dong/NetworkWordCount.jar dong.spark.SocketSimulation/
home/dong/Streamingtest/file1.txt 9999 1000
```

数据流模拟器每间隔 1000 毫秒从/home/dong/Streamingtext/file1.txt 中随机截取一行文本发送给 master 节点的 9999 端口。在分析器未连接时，数据流模拟器处于阻塞状态，终端不会显示输出的文本。

第 5 步，运行分析器。

在 master 上启动 IntelliJ IDEA 编写分析器代码，然后单击菜单"Build->"Build Artifacts"，通过 Application 选项配置分析器运行所需的参数，其中 Socket 主机名为 master、端口号为 9999，参数之间用空格间隔，如图 7.20 所示。

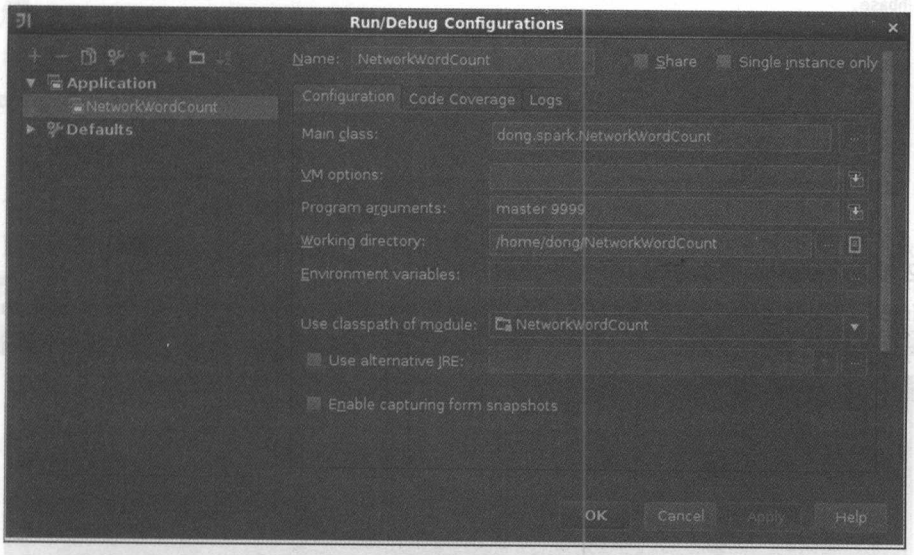


图 7.20　分析器参数配置示意图

配置好参数后返回 IntelliJ IDEA 菜单栏，单击"Run"->"Build Artifacts"运行分析器。

4．查看结果

第 1 步，在 master 上查看数据流模拟器运行情况。

IntelliJ IDEA 运行分析器从而与数据流模拟器建立连接。当检测到外部连接时，数据流模拟器将每隔 1000 毫秒从/home/dong/Streamingtext/file1.txt 中随机截取一行文本发送给 master 节点上的 9999 端口。为方便讲解和说明，file1.txt 中每一行只包含一个单词，因此数据流模拟器每次仅发送一个单词给端口，如图 7.21 所示。

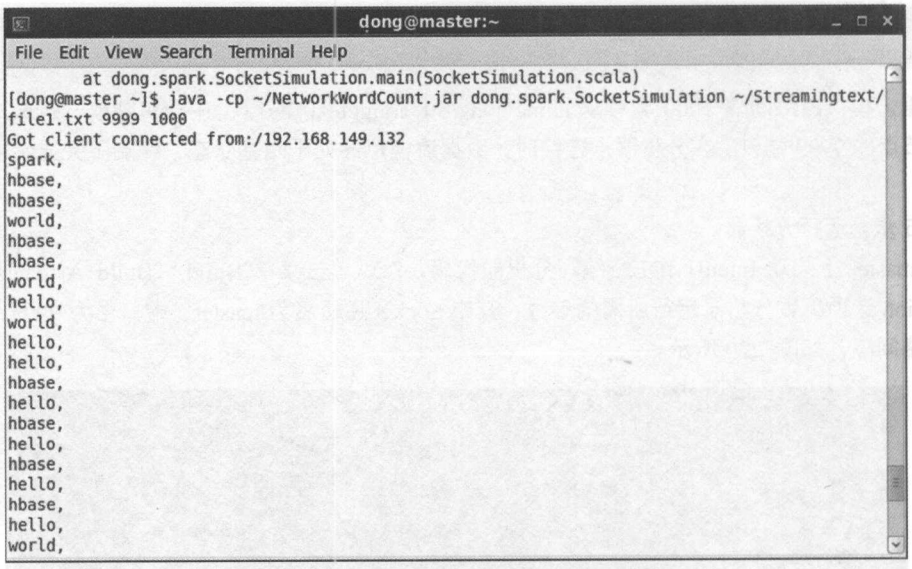


图 7.21　在 master 上模拟器运行结果

第 2 步，在 master 的 IntelliJ IDEA 中查看分析器运行情况。

在 IntelliJ IDEA 的运行日志窗口中，可以观察到统计结果。通过分析可知 Spark Streaming 每个批处理时间间隔内获取的单词数为 5，刚好是 5 秒内发送单词的总数，并对各单词进行了统计，如图 7.22 所示。

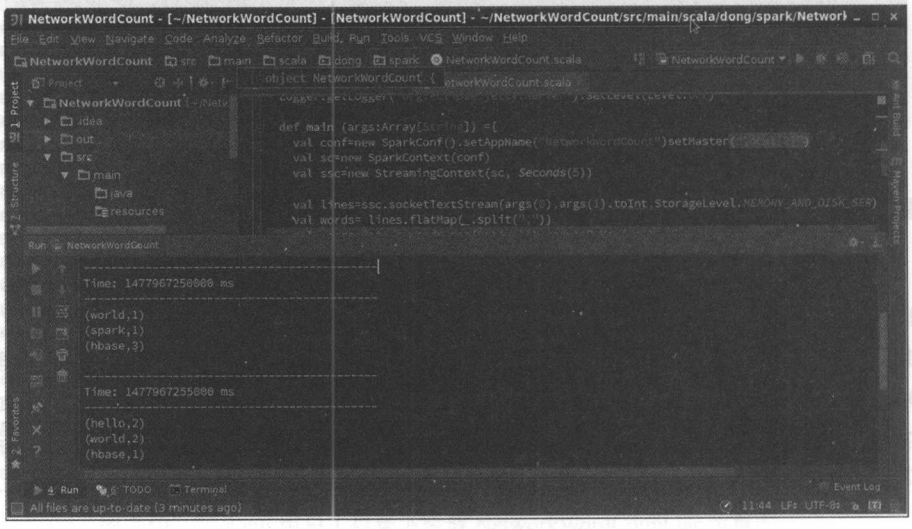


图 7.22　IntelliJ IDEA 中分析器运行结果

7.6.5 网络数据处理案例（二）

1. 功能需求

监听网络中非本地节点指定端口传输的数据（本例监听 slave1 节点上 9999 端口的英文文本数据，以逗号间隔单词），每 5 秒统计一次该时间间隔内收集到各单词的个数。

2. 代码实现

本例功能的实现同样涉及数据流模拟器和分析器两部分。

数据流模拟器沿用 7.6.4 节中数据流模拟器代码，无需修改。但是运行时与 7.6.4 不同，本例将在 slave1 节点启动数据流模拟器，用以模拟 slave1 节点上 9999 端口传输的数据；将在 master 节点以集群方式启动分析器。

分析器代码沿用 7.6.4 节中分析器代码，但需要将 setMaster("local[2]") 修改为 setMaster("spark://192.168.149.132:7077")。

3. 运行演示

第 1 步，在 slave1 节点启动数据流模拟器。

（1）将 7.6.4 节中案例的 master 节点上创建的 Streamingtext 文件夹以及其中的 file1.txt 复制至 slave1 节点 dong 用户主目录下，如图 7.23 所示。

```
$ scp -r /home/dong/Streamingtext slave1:~/
```

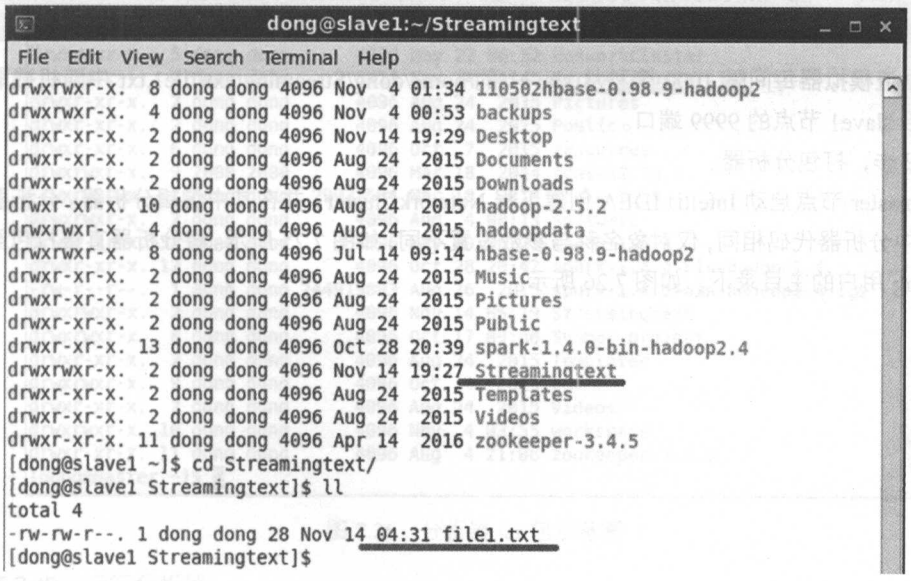


图 7.23 slave1 上 Streamingtext 目录以及其中的 file1.txt 示意图

（2）将 7.6.4 节案例中 master 节点上的数据流模拟器 NetworkWordCount.jar 复制至 slave1 节点

dong 用户的主目录下，如图 7.24 所示。

```
$ scp -r /home/dong/NetworkWordCount.jar slave1:~/
```

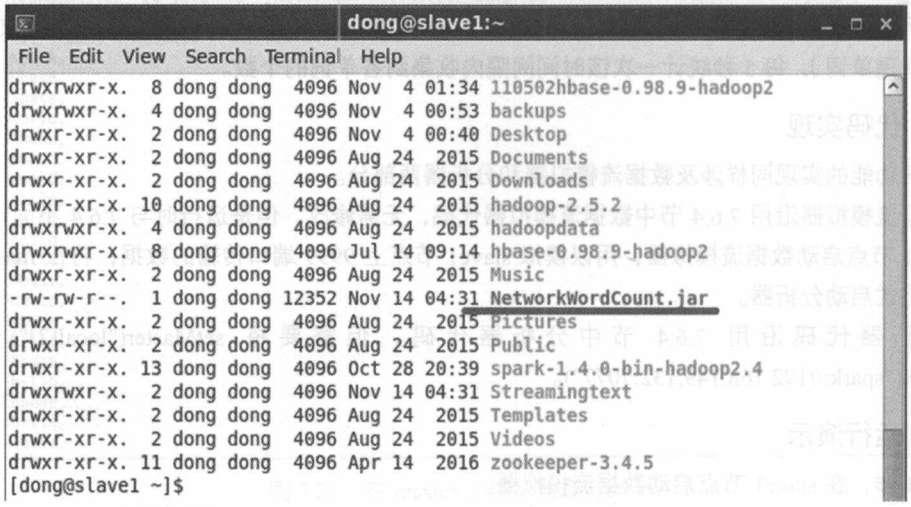


图 7.24　slave1 中目录下 NetworkWordCount.jar 示意图

（3）在 slave1 节点开启终端，通过下面代码启动数据流模拟器。

```
$ java -cp NetworkWordCount.jar dong.spark.SocketSimulation/home/dong/
Streamingtext/file1.txt 9999 1000
```

数据流模拟器每间隔 1000 毫秒从 slave1 的/home/dong/Streamingtext/file1.txt 中随机截取一行文本发送给 slave1 节点的 9999 端口。

第 2 步，打包分析器。

在 master 节点启动 IntelliJ IDEA 创建工程 NetworkCluster，并在其中编辑分析器。分析器代码与 7.6.4 节中分析器代码相同，仅对象名称与参数设置不同，如图 7.25 所示。将分析器直接打包至 master 节点 dong 用户的主目录下，如图 7.26 所示。

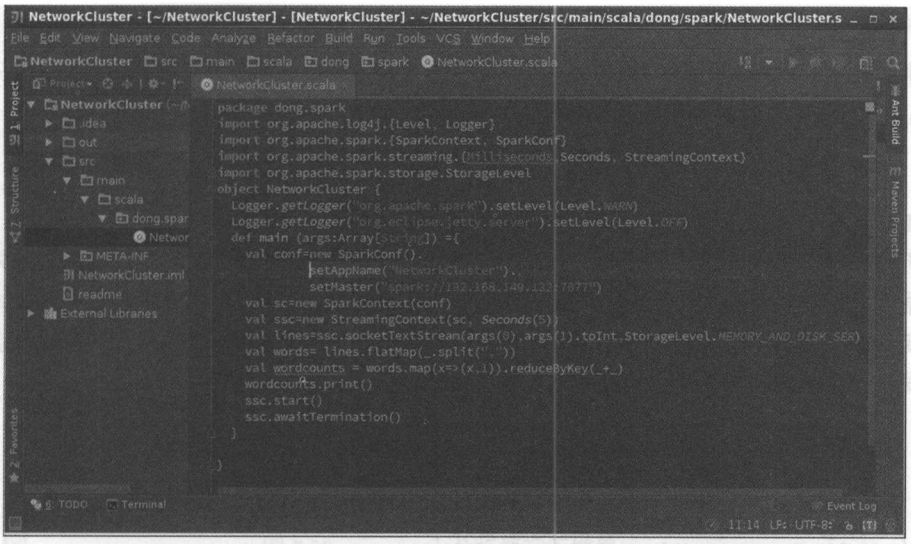


图 7.25 IntelliJ IDEA 中分析器示意图

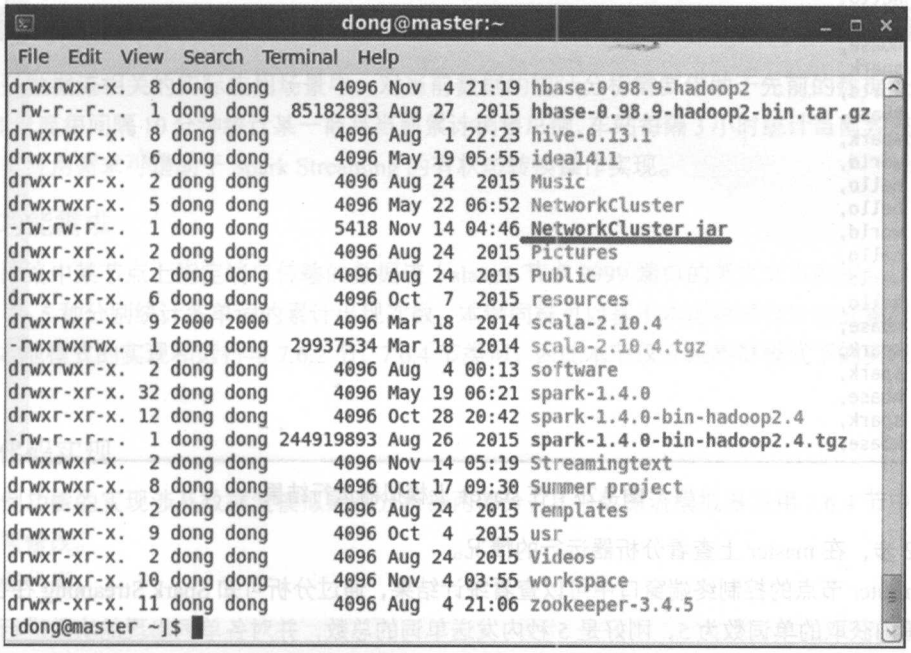


图 7.26 分析器 jar 包示意图

第 3 步，运行分析器。

在 master 节点开启终端，进入 spark-1.4.0-bin-hadoop2.4 文件夹执行下面命令向 Spark 集群提交应用程序。

```
$ bin/spark-submit ~/NetworkCluster.jar slave1 9999
```

其中“slave1”为分析器中参数 args(0)的取值，“9999”为分析器中 args(1)的取值，分别表示 Streaming 应用程序监听的主机名和端口号。

4. 查看结果

第 1 步，在 slave1 上查看数据流模拟器运行情况

分析器在集群上提交运行后便与 slave1 上运行的数据流模拟器建立起连接。当检测到外部连接时，数据流模拟器将每隔 1000 毫秒从/home/dong/Streamingtext/file1.txt 中随机截取一行文本发送给 slave1 节点的 9999 端口。由于该文本文件中每一行只包含一个单词和一个逗号，因此数据流模拟器每次仅发送一个单词和一个逗号给端口，如图 7.27 所示。

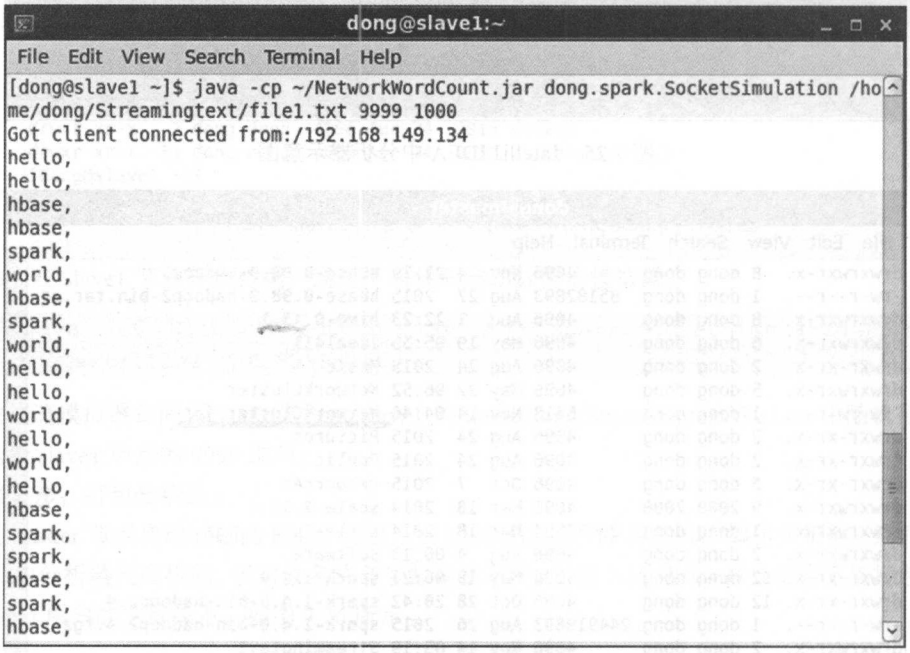


图 7.27　在 slave1 上模拟器运行结果

第 2 步，在 master 上查看分析器运行的情况。

在 master 节点的控制终端窗口中可以查看统计结果，通过分析可知 Spark Streaming 在各批处理时间间隔内获取的单词数为 5，刚好是 5 秒内发送单词的总数，并对各单词出现的次数进行了统计，如图 7.28 所示。

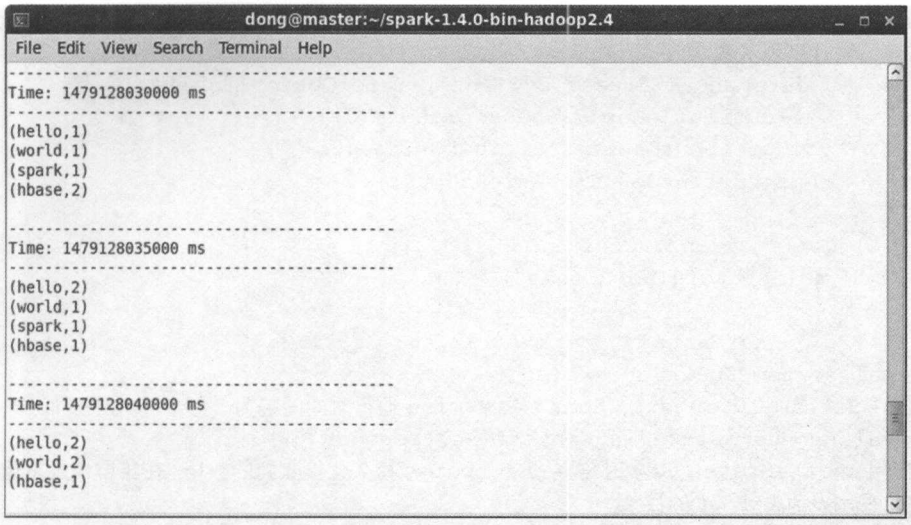


图 7.28 master 上分析器运行结果

7.6.6 stateful 应用案例

在很多数据流相关的实际应用场景中，对当前数据的统计分析需要借助于先前的数据处理结果完成。例如电商每间隔 10 分钟统计某一商品当前累计销售总额、车站每隔 3 小时统计当前客流总量，等等。此类应用需求可借助于 Spark Streaming 的有状态转换操作实现。

1. 功能需求

监听网络中某节点上指定端口传输的数据流（slave1 节点 9999 端口的英文文本数据，以逗号间隔单词），每 5 秒分别统计各单词的累计出现次数。本例同样可以基于本地模式和分布式集群模式实现，其中本地模式的实现和运行与 7.6.2 节、7.6.4 节类似，因此本节仅介绍集群模式下的实现与运行过程。

2. 代码实现

本案例功能的实现涉及数据流模拟器和分析器两部分，其中数据流模拟器沿用 7.6.4 节中模拟器代码，无需修改。

分析器代码：

```
package dong.spark
import org.apache.spark.{SparkContext, SparkConf}
import                org.apache.spark.streaming.{Milliseconds,Seconds,
StreamingContext}
import org.apache.spark.streaming.StreamingContext._
object StatefulWordCount {
  def main(args:Array[String]): Unit ={
```

```
    /*定义更新状态方法，参数 values 为当前批处理时间间隔内各单词出现的次数，state 为以往所有批次各单词累计出现次数。*/
        val updateFunc=(values: Seq[Int],state:Option[Int])=>{
            val currentCount=values.foldLeft(0)(_+_)
            val previousCount=state.getOrElse(0)
            Some(currentCount+previousCount)
        }
        val conf=new SparkConf().
    setAppName("StatefulWordCount").

                setMaster("spark://192.168.149.132:7077")
        val sc=new SparkContext(conf)
    //创建 StreamingContext，Spark Steaming 运行时间间隔为 5 秒。
        val ssc=new StreamingContext(sc, Seconds(5))
    /*使用 updateStateByKey 时需要 checkpoint 持久化接收到的数据。在集群模式下运行时，需要将持久化目录设为 HDFS 上的目录。*/
         ssc.checkpoint("hdfs://master:9000/user/dong/input/StatefulWordCount
log")
    /*通过 Socket 获取指定节点指定端口的数据创建 DStream,其中节点与端口分别由参数 args(0)和 args(1)给出。*/
        val lines=ssc.socketTextStream(args(0),args(1).toInt)
        val words=lines.flatMap(_.split(","))
        val wordcounts=words.map(x=>(x,1))
    //使用 updateStateByKey 来更新状态，统计从运行开始以来单词总的次数。
        val stateDstream=wordcounts.updateStateByKey[Int](updateFunc)
        stateDstream.print()
        ssc.start()
        ssc.awaitTermination()
      }
    }
```

3. 运行演示

第 1 步，slave1 节点启动数据流模拟器。

方法与过程与 7.6.5 节中相同。

第 2 步，打包分析器。

master 节点启动 IntelliJ IDEA 创建工程 StatefulWordCount 编辑分析器，如图 7.29 所示，并将分析器直接打包至 master 节点 dong 用户的主目录下，如图 7.30 所示。

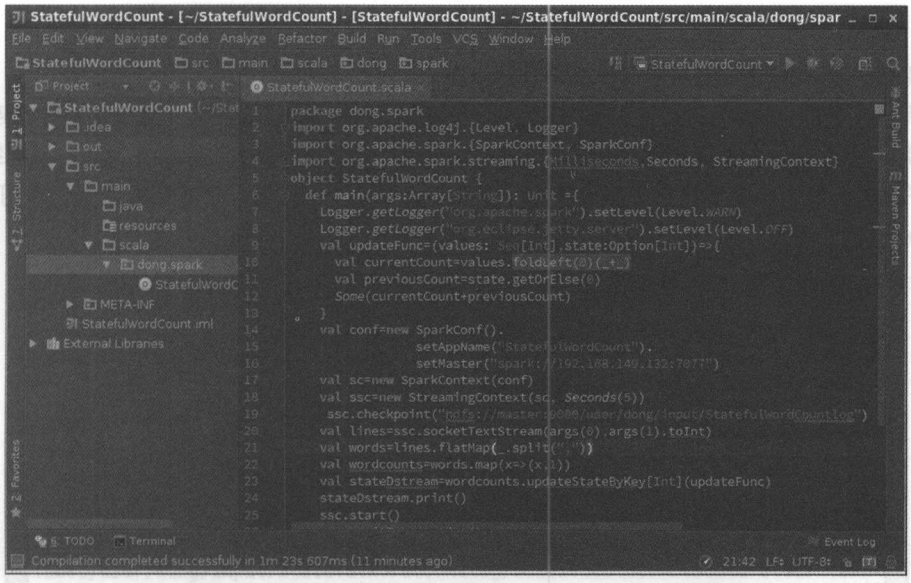


图 7.29 IntelliJ IDEA 中 StatefulWordCount 示意图

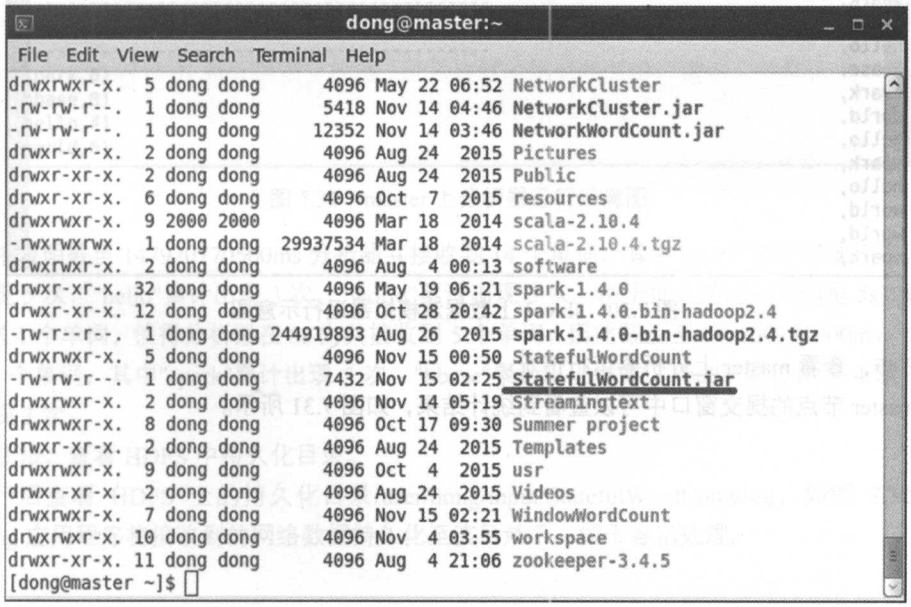


图 7.30 master 上的 StatefulWordCount.jar 示意图

第 3 步，运行分析器。

在 master 节点开启终端，通过下面代码向 Spark 集群提交应用程序。

```
$ bin/spark-submit ~/StatefulWordCount.jar slave1 9999
```

4. 查看结果

第 1 步，查看 slave1 上数据流模拟器运行情况

分析器在集群上提交运行后与 slave1 上运行的数据流模拟器建立连接。当检测到外部连接时，数据流模拟器将每隔 1000 毫秒从/home/dong/Streamingtext/file1.txt 中随机截取一行文本发送给 slave1 节点上的 9999 端口。由于该文本文件中每一行只包含一个单词，因此每秒仅发送一个单词给端口。如图 7.30 所示。

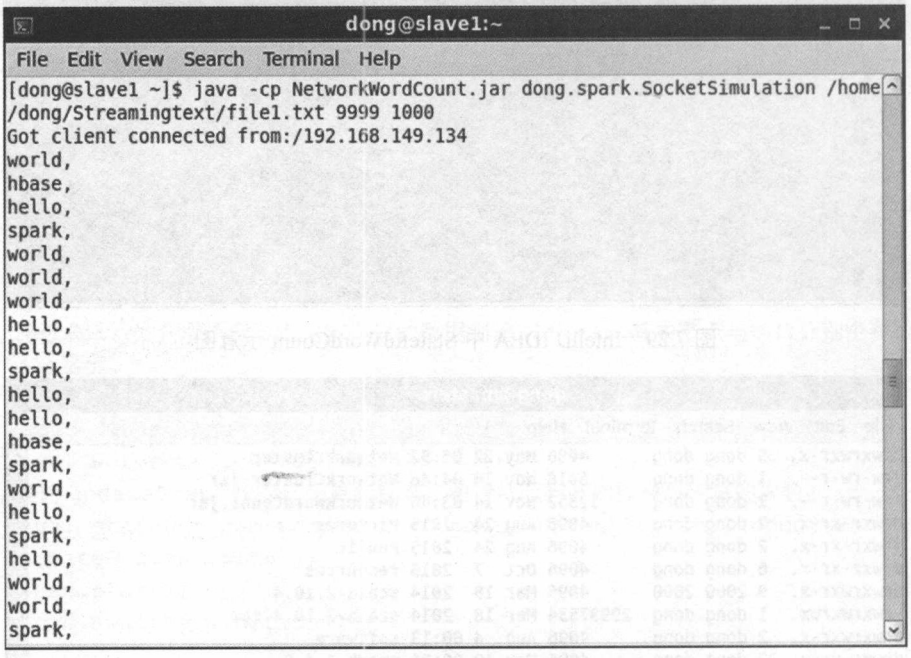


图 7.30　slave1 上数据流模拟器运行示意图

第 2 步，查看 master 上分析器运行情况。

在 master 节点的提交窗口中可以查看到统计结果，如图 7.31 所示。

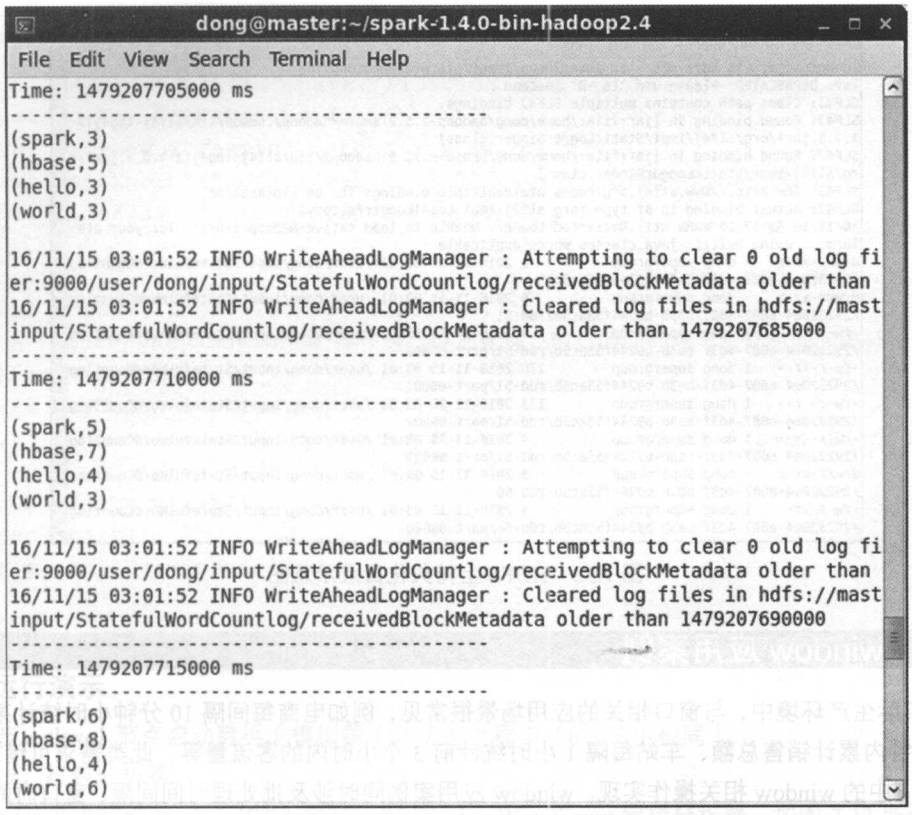


图 7.31 master 上分析器运行示意图

图中表明截至 147920770500ms 分析器共接收到 14 个单词，其中"spark"累计出现 3 次，"hbase"累计出现 5 次，"hello"累计出现 3 次，"world"累计出现 3 次。由于批处理时间间隔是 5s，模拟器每 1 秒发送 1 个单词，使得分析器在 5s 内共接收到 5 个单词，因此截止至 147920771000ms，分析器共收到 19 个单词，其中"spark"累计出现 5 次，"hbase"累计出现 7 次，"hello"累计出现 4 次，"world"累计出现 3 次。

第 3 步，查看 HDFS 中持久化目录。

运行后查看 HDFS 上的持久化目录/user/dong/input/StatefulWordCountlog，如图 7.32 所示。Streaming 应用程序将接收到的网络数据持久化至该目录下，便于容错处理。

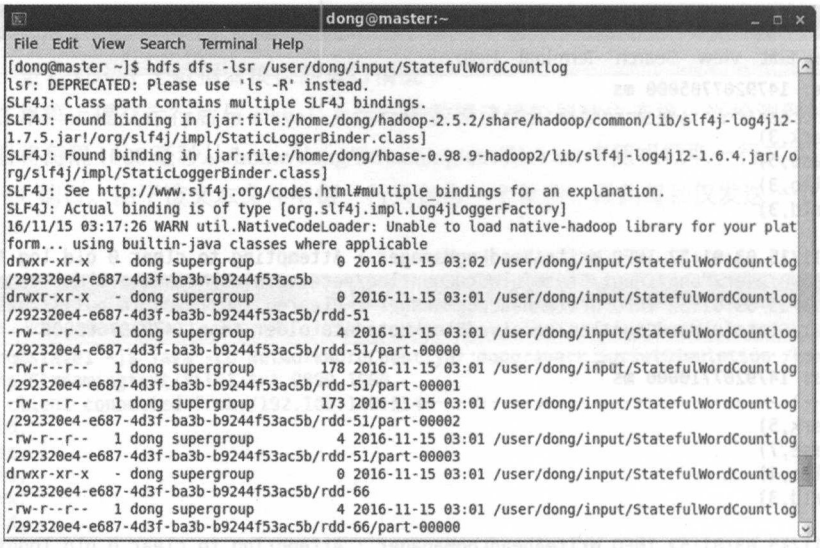


图 7.32　HDFS 上持久化目录示意图

7.6.7　window 应用案例

在实际生产环境中，与窗口相关的应用场景很常见，例如电商每间隔 10 分钟小时统计某一商品前 30 分钟内累计销售总额、车站每隔 1 小时统计前 3 个小时内的客流量等，此类需求可借助 Spark Streaming 中的 window 相关操作实现。window 应用案例同时涉及批处理时间间隔、窗口时间间隔与滑动时间间隔。本节 window 应用案例可视为是对 7.6.6 节中案例的进一步扩展。

1．功能需求

监听网络中某节点上指定端口传输的数据流（slave1 节点上 9999 端口的英文文本数据，以逗号间隔单词），每 10 秒统计前 30 秒各单词累计出现的次数。本例仅介绍集群模式下的实现过程，本地模式的实现和运行与 7.6.2 节、7.6.4 节类似。

2．代码实现

本例功能的实现涉及数据流模拟器和分析器两部分。数据流模拟器沿用 7.6.5 节中模拟器代码，无需修改。

分析器代码：

```
package dong.spark
import org.apache.spark.{SparkContext, SparkConf}
import org.apache.spark.streaming.StreamingContext._
import org.apache.spark.streaming._
import org.apache.spark.storage.StorageLevel
object WindowWordCount {
```

```
    def main(args:Array[String]) ={
      val conf=new SparkConf().setAppName("WindowWordCount").
               setMaster("spark://192.168.149.132:7077")
      val sc=new SparkContext(conf)
      val ssc=new StreamingContext(sc, Seconds(5))
      ssc.checkpoint("hdfs://master:9000/user/dong/WindowWordCountlog")
      val lines=ssc.socketTextStream( args(0),
                                      args(1).toInt,
                                      StorageLevel.MEMORY_ONLY_SER)
      val words= lines.flatMap(_.split(","))
    /*采用 reduceByKeyAndWindow 操作进行叠加处理，窗口时间间隔与滑动时间间隔分别由参数
args(2)和 args(3)给出。*/
      val wordcounts=words.map(x=>(x,1)).
                     reduceByKeyAndWindow((a:Int,b:Int)=>(a+b),Seconds(ar
                     gs(2).toInt),Seconds(args(3).toInt))
      wordcounts.print()
      ssc.start()
      ssc.awaitTermination()
    }
  }
```

3. 运行演示

第 1 步，slave1 节点启动数据流模拟器（方法与过程与 7.6.5 节中相同）。

第 2 步，打包分析器。

在 master 节点启动 IntelliJ IDEA 创建工程 WindowWordCount 编辑分析器，如图 7.33 所示，并将分析器直接打包至 master 节点 dong 用户的主目录下，如图 7.34 所示。

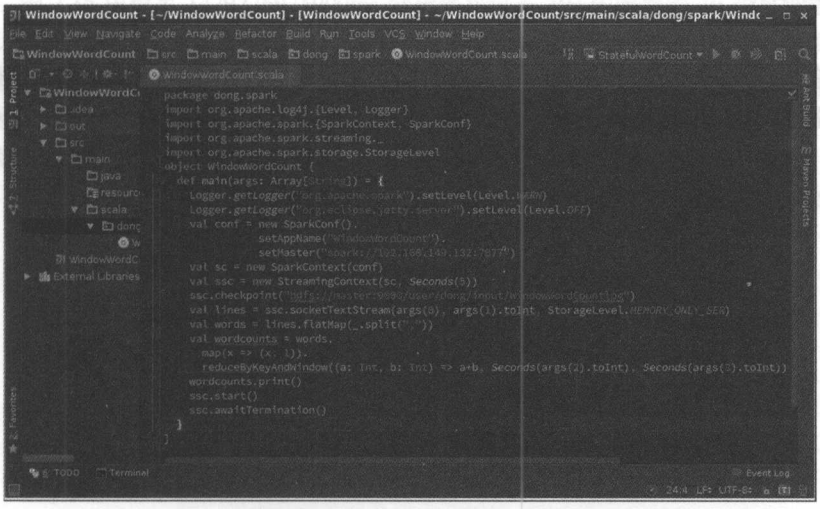


图 7.33 IntelliJ IDEA 中 WindowWordCount 示意图

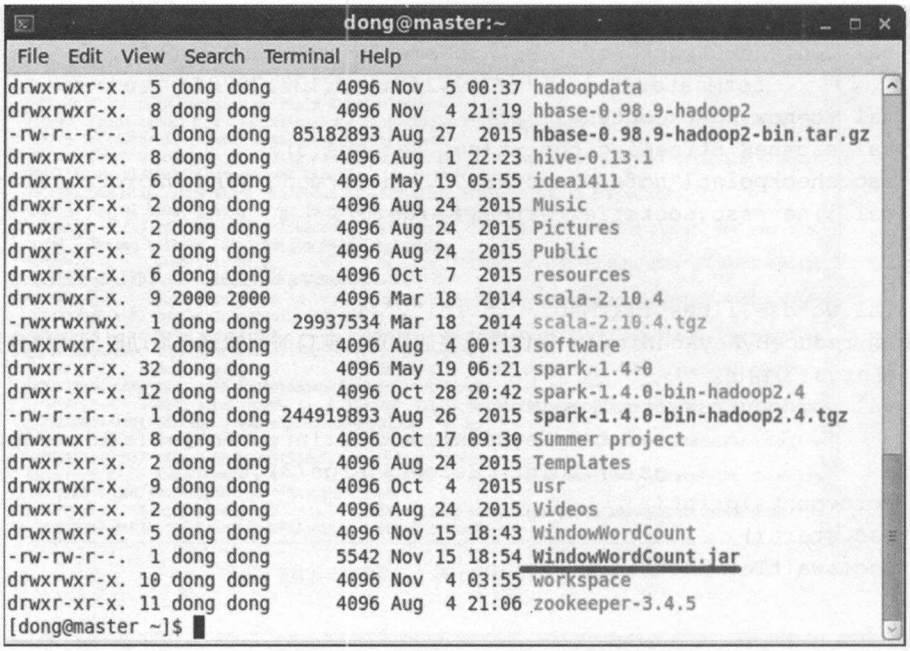


图 7.34　master 上 WindowWordCount.jar 示意图

第 3 步，运行分析器。

在 master 节点开启终端，通过下面代码向 Spark 集群提交应用程序。

```
$ bin/spark-submit ~/WindowWordCount.jar slave1 9999 30 10
```

4. 查看结果

第 1 步　在 slave1 上查看数据流模拟器运行情况。

分析器在集群上提交运行后与 slave1 上运行的数据流模拟器建立连接。当检测到外部连接时，数据流模拟器将每隔 1000 毫秒从/home/dong/Streamingtext/file1.txt 中随机截取一行文本发送给 slave1 节点的 9999 端口。由于该文本文件中每一行只包含一个单词和一个逗号，因此每秒仅发送一个单词和一个逗号给端口，如图 7.35 所示。

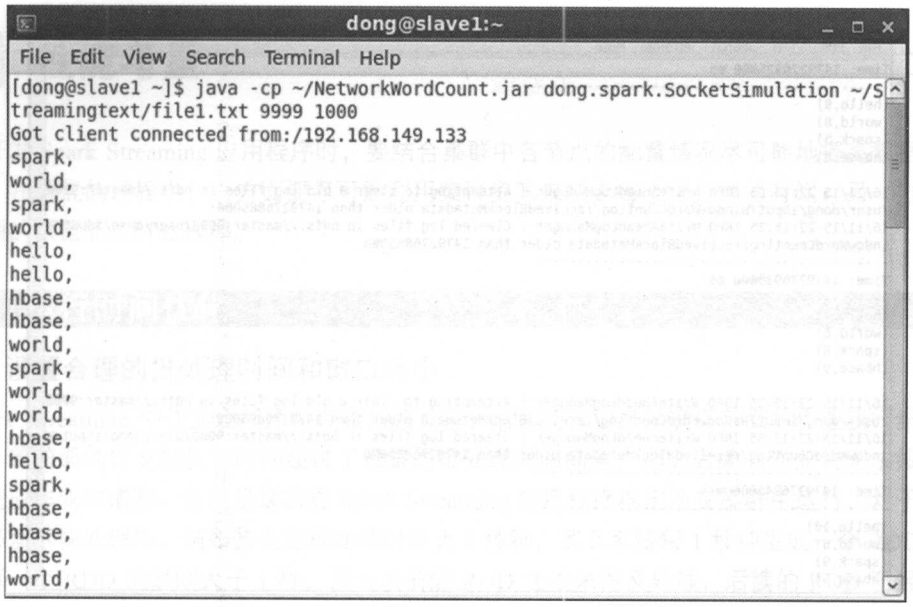


图 7.35 在 slave1 上数据流模拟器运行示意图

第 2 步，在 master 上查看分析器运行情况。

在 master 节点的提交窗口中可以查看到统计结果。在 WindowWordCount 应用程序启动初期，窗口并没有被接收到的单词填满，但随着时间的推进，每个窗口中的单词数目最终固定为 30 个。图 7.35 只是截取了运行结果中的三个批次。由于设定了窗口时间间隔是 30s，滑动时间间隔是 10s，且数据流模拟器每间隔 1s 发送一个单词，因此 WindowWordCount 每间隔 10s 对过去 30s 内收到的各单词个数进行统计。图 7.36 中截至 1479276925000ms 分析器对过去 30s 内收到的 30 个单词进行统计，其中"spark"累计出现 5 次，"hbase"累计出现 8 次，"hello"累计出现 9 次，"world"累计出现 8 次。再间隔 10s，截至 1479276935000ms，分析器对过去 30s 内收到的 30 个单词进行统计，其中"spark"累计出现 8 次，"hbase"累计出现 9 次，"hello"累计出现 7 次，"world"累计出现 6 次。

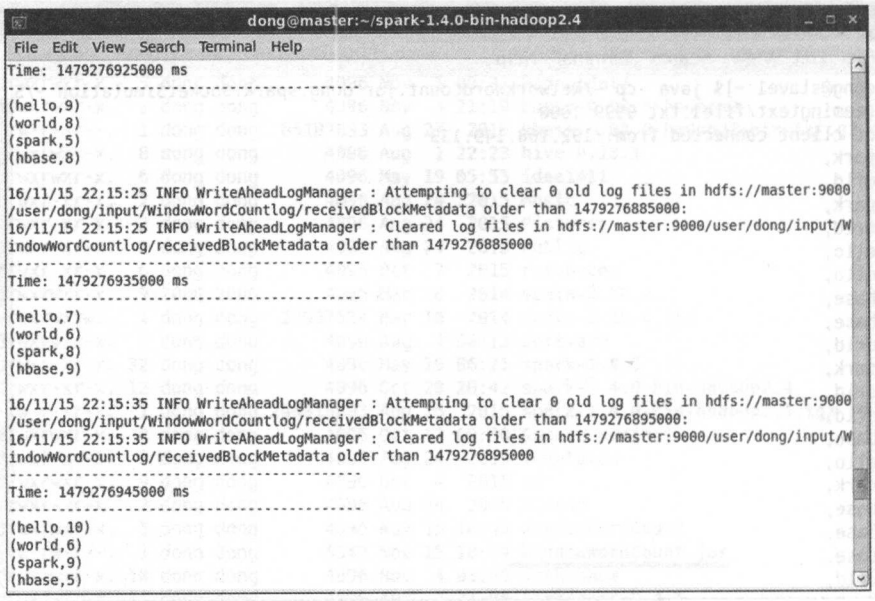


图 7.36　在 master 上分析器运行示意图

第 3 步，查看持久化数据。

运行后查看 HDFS 上的持久化目录/user/dong/input/WindowWordCountlog，如图 7.37 所示。Streaming 应用程序将接收到的网络数据持久化至该目录下，便于容错处理。

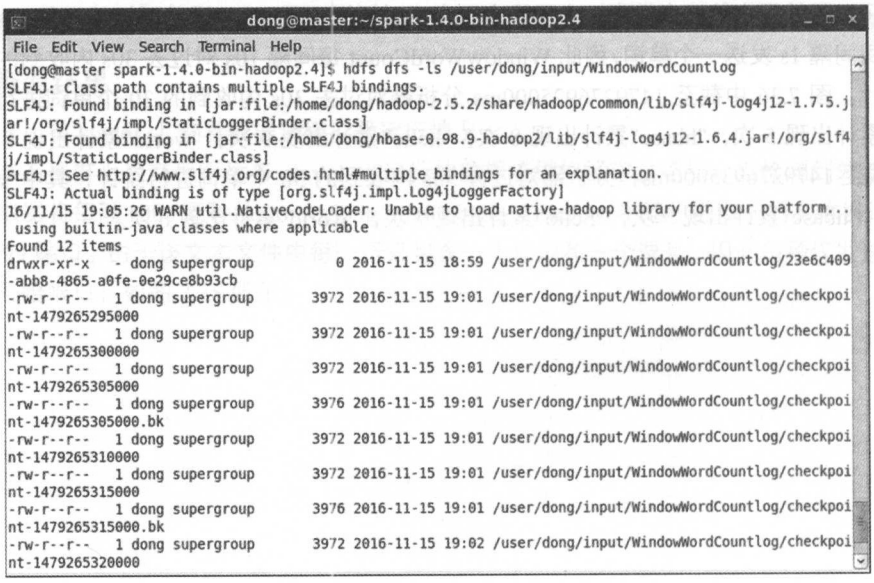


图 7.37　HDFS 上持久化目录示意图

7.7 性能考量

在开发 Spark Streaming 应用程序时，要结合集群中各节点的配置情况尽可能地提高数据处理的实时性。在调优的过程中，一方面要尽可能利用集群资源来减少每个批处理的时间；另一方面要确保接收到的数据能及时处理掉。

7.7.1 运行时间优化

1. 设置合理的批处理时间和窗口大小

Spark Streaming 中作业之间通常存在依赖关系，后面的作业必须确保前面的作业执行结束后才能提交，若前面的作业的执行时间超过了设置的批处理时间间隔，那么后续的作业将无法按时提交执行，造成作业的堵塞。也就是说若想 Spark Streaming 应用程序稳定地在集群中运行，对于接收到的数据必须尽快处理掉。例如若设定批处理时间为 1 秒钟，那么系统每 1 秒钟生成一个 RDD，如果系统计算一个 RDD 的时间大于 1 秒，那么当前的 RDD 还没来得及处理，后续的 RDD 已经提交上来在等待处理了，这就产生了堵塞。因此需要设置一个合理的批处理时间间隔以确保作业能够在这个批处理时间间隔时间内结束。许多实验数据表明，500 毫秒对大多 Spark Streaming 应用而言是较好的批处理时间间隔。

类似地，对于窗口操作，滑动时间间隔对于性能也有很大的影响。当单批次数据计算代价过高时，可以考虑适当增大滑动时间间隔。

对于批处理时间和窗口大小的设定，并没有统一的标准。通常是先从一个比较大的批处理时间（10 秒左右）开始，然后不断地使用更小的值进行对比测试。如果 Spark Streaming 用户界面中显示的处理时间保持不变，则可以进一步设定更小的值；如果处理时间开始增加，则可能已经达到了应用的极限，再减小该值则可能会影响系统的性能。

2. 提高并行度

提高并行度也是一种减少批处理所消耗时间的常见方法。有以下三种方式可以提高并行度。一种方法是增加接收器数目。如果获取的数据太多，则可能导致单个节点来不及对数据进行读入与分发，使得接收器成为系统瓶颈。这时可以通过创建多个输入 DStream 来增加接收器数目，然后再使用 union 来把数据合并为一个数据源。第二种方法是将收到的数据显式地重新分区。如果接收器数目无法再增加，可以通过使用 DStream.repartition、spark.streaming.blocklnterval 等参数显式地对 Dstream 进行重新分区。第三种方法是提高聚合计算的并行度。对于会导致 shuffle 的操作，例如 reduceByKey、reduceByKeyAndWindow 等操作，可通过显示设置更高的行度参数确保更为充分地使用集群资源。

7.7.2 内存使用与垃圾回收

1. 控制批处理时间间隔内的数据量

Spark Streaming 会把批处理时间间隔内获取到的所有数据存放在 Spark 内部可用的内存中。因此必须确保在当前节点上 SparkStreaming 可用的内存容量至少能容下一个批处理时间间隔内所有的数据。比如一个批处理时间间隔是 1 秒，但是 1 秒产生了 1GB 的数据，那么要确保当前的节点上至少有可供 SparkStreaming 使用的 1GB 内存。

2. 及时清理不再使用的数据

对于内存中处理过的、不再需要的数据应及时清理，以确保 Spark Streaming 能够拥有足够的内存空间可以使用。一种方法是可以通过设置合理的 spark.cleaner.ttl 时长来及时清理超时的无用数据，但该方法应慎重使用，以免后续数据在需要时被错误清理。另一种方法是将 spark.streaming.unpersist 设置为 true，系统将自动清理已经不需要的 RDD。该方法能显著减少 RDD 对内存的需要，同时潜在地提高 GC 的性能。此外用户还可以通过配置参数 streamingContext.remember 为数据设置更长的保留时间。

3. 减少序列化与反序列化的负担

SparkStreaming 默认将接收到的数据序列化后放入内存，以减少内存使用。序列化和反序列化需要更多的 CPU 资源，因此使用适当的序列化工具（例如 Kryo）和自定义的序列化接口可以更高效地使用 CPU。除了使用更好的序列化工具外还可以结合压缩机制，通过配置 spark.rdd.compress，以 CPU 的时间开销来换取内存资源，降低 GC 开销。

第8章

Spark MLlib

8.1 Spark MLlib 概述

8.1.1 机器学习介绍

在人类的诸多行为中，学习行为是最为重要的一种行为。机器学习（Machine Learning，ML）重点研究如何让计算机模拟人类的学习行为，从而获取新的知识和技能、改善具体算法的性能；即如何通过以往的经验或数据自动改进的计算机算法。机器学习是一门多领域交叉学科，涉及概率论、统计学、逼近论、凸分析等多门学科。机器学习是人工智能的核心，是使计算机具有智能的根本途径，其应用遍及人工智能的各个领域。

对于给定的一个训练数据集，机器学习的过程是通过不断地分析和学习得到一个分类函数（分类模型）或预测函数（预测模型）。通过这个模型可以对输入对象进行分类或对对象的特征向量进行预测。

机器学习过程强调三个关键词：任务、经验、性能，如图 8.1 所示。首先设计算法，基于训练数据利用该算法构建出机器学习模型（分类模型或预测模型）；然后利用测试数据对模型进行评估。如

果评估的性能达到要求就用这个模型来对其他数据进行分类或预测，如果达不到要求则调整算法重新构建模型。重复这一过程，直至得到满意的模型为止。

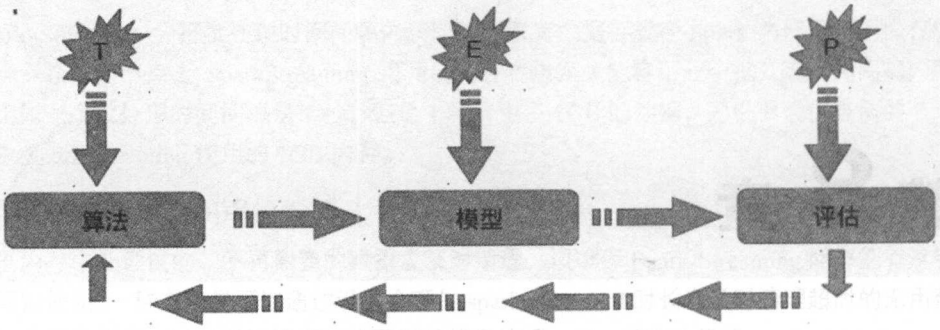


图 8.1　机器学习流程图

机器学习问题可大致分为以下几类：

1. 监督学习

监督学习通常用于解决分类和数值预测问题。在监督学习中，训练集中的每个数据包括输入和输出两部分，也称为特征和目标。训练集中的目标是由人事先标注给出的（例如垃圾邮件，正常邮件；优、良、中、差）。训练集中所有目标构成目标系统，目标系统又分为二元目标系统（垃圾、正常邮件）和多元目标系统（优、良、中、差）。在建立预测模型后，监督学习建立一个学习过程，将数据对象的预测结果与其实际结果进行对比，不断调整预测模型，直到模型的预测结果满足预期的阈值范围。常见的监督学习算法有回归分析和统计分类两种。

2. 无监督学习

与监督学习相比，无监督学习没有预先标注好的目标系统，因此机器学习算法需要数据的属性和结构信息。常见的无监督学习算法有聚类和关联规则两种。这类机器学习问题的目标不是让效用函数最大化，而是找到训练数据中的近似点。聚类常常能发现那些与假设匹配的相当好的直观分类，例如基于人口统计的聚合个体可能会在一个群体中形成一个富有的聚合，以及其他的贫穷的聚合。

3. 半监督学习

半监督学习是介于监督学习和无监督学习之间的一种机器学习方式，是机器学习领域研究的一个重点问题。它主要考虑如何利用少量的标注样本和大量的未标注样本进行学习。半监督学习对于减少标注代价，提高学习机器的性能具有非常重要的意义。概率方法、图论方法、支持向量机等都是处理半监督学习问题常用的方法。半监督学习分类算法提出的时间比较短，还有许多方面尚有待深入的研究。

4. 强化学习

强化学习又称为再励学习、评价学习，是智能系统从环境到行为映射的学习，以使奖励信号（强

化信号）函数值最大。在强化学习模式下，输入数据直接反馈到模型，模型必须立即作出调整。强化学习是重要的一类机器学习问题，在智能控制机器人及分析预测等领域有许多应用。

8.1.2 Spark MLlib 简介

SparkMLlib（Machine Learnig lib）是常用的机器学习算法的 Spark 实现库，同时包括相关的测试和数据生成器。机器学习算法通常涉及较多的迭代计算，而 Spark 的设计初衷正是为了高效地处理迭代式作业。作为 Spark 上的机器学习组件，MLlib 继承了 Spark 先进的内存存储模式和作业调度策略，使得其对机器学习问题的处理速度大大高于普通的数据处理引擎，主要特性如下。

1. 速度快

机器学习算法通常需要在多次迭代后获得足够小的误差或者足够收敛才会停止。如果使用 MapReduce 计算框架进行迭代式计算，每次计算都要读写磁盘，这会导致非常大的 I/O 和 CPU 开销。而 Spark 是基于内存的运算，迭代的中间结果缓存在内存中，因此处理速度会有大幅提升。

2. 易用

用户不仅可以直接使用 MLlib 中提供的经典机器学习算法 API，还可以基于 RDD 中封装的丰富操作开发自己的机器学习算法。因此使用和开发基于 Spark 的机器学习算法非常方便简洁。

3. 集成度高

Spark 基于 RDD 构建起来的 MLlib 可以和 Spark 中的 Spark SQL、GraphX、Spark Streaming 等其他组件进行无缝连接。例如可以借助 MLlib 对 Spark Streaming 接收到的实时数据流进行训练，这对一些复杂实时流计算场景非常有价值；MLlib 也可以与 GraphX 相结合进行深度的机器学习；MLlib 还可以直接对 Spark SQL 查询到的数据进行分析。

MLlib 目前支持 4 种常见的机器学习问题：分类算法、回归算法、聚类和协同过滤，其架构如图 8.2 所示。

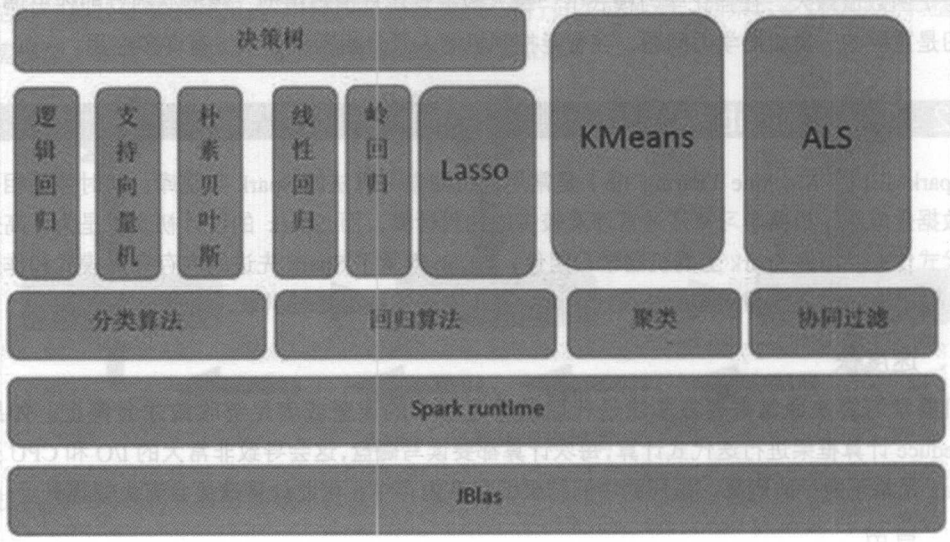


图 8.2　MLlib 架构

MLlib 机器学习库还在持续地更新中，Apache 的相关研究人员仍在不断地为其中添加更多的机器学习算法。

8.2 MLlib 向量与矩阵

在设计和实现机器学习算法的过程中，很多问题都是借助矩阵这一数学工具来解决的。所以向量、矩阵在 MLlib 中的表示与存储方式以及其上的相关操作非常重要。

8.2.1 MLlib 向量

1．密集向量和稀疏向量

向量（Vector）是 MLlib 中最常用的数据类型，具体可分为密集向量和稀疏向量两种。密集向量把所有维度的值存放在一个浮点型数组中。相比之下，稀疏向量只把各维度中的非 0 元素存储下来。对于密集向量与稀疏向量中的非 0 元素个数，并没有严格的定义与要求。通常当向量中最多不超过 10%的元素为非 0 元素时，更倾向于将其定义为稀疏向量（不仅是出于对内存使用的考虑，也是出于对速度的考虑）。在 MLlib 中向量对应 Vector 类型，它是 Spark MLlib1.0 后提出的一种数据结构。Vector 类型极大地方便了 MLlib 对数据的描述和操作。定义密集向量可直接采用 Vector 中的 dense 方法，定义稀疏向量可采用 Vector 中的 sparse 方法。

在 Spark-shell 下演示示例 8-1~示例 8-5，首先打开 Spark-shell 并引入相应的 jar 包：

```
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.regression.LabeledPoint
```

示例 8-1

```
val dv: Vector = Vectors.dense(1.0, 0.0, 3.0)
```

实例 8-1 将(1.0, 0.0, 3.0)创建为密集向量。

运行结果：

```
dv: org.apache.spark.mllib.linalg.Vector = [1.0,0.0,3.0]
```

示例 8-2

```
val sv1: Vector = Vectors.sparse(5, Array(0, 2), Array(1.0, 3.0))
```

运行结果：

```
sv1: org.apache.spark.mllib.linalg.Vector = (5,[0,2],[1.0,3.0])
```

实例 8-2 将(1.0, 0.0, 3.0, 0.0, 0.0)创建为稀疏向量。由于稀疏向量中 0 多非 0 少，因此在创建时仅需指出非 0 值的索引项和值。sv1 表示向量中一共有 5 个元素，其中第一个数组表示非 0 元素的索引值，即索引为 0 和 2 的元素为非 0 元素；第二个数组表示非 0 元素的具体值为 1.0 和 3.0。除此之外，还可以借助序列更为直观地创建稀疏向量。

示例 8-3

```
val sv2: Vector = Vectors.sparse(5, Seq((0, 1.0), (2, 3.0)))
```

运行结果：

```
val sv2: Vector = Vectors.sparse(5, Seq((0, 1.0), (2, 3.0)))
```

示例 8-3 在创建 sv2 时用序列来指定向量中的每一个非 0 元素，将所有非 0 元素的索引和值构成一个二元组放在序列中。

需要注意，MLlib 中所有的向量都是本地向量。

2. LabeledPoint

LabeledPoint 是一种基于 Vector 扩展得到的数据类型，同样仅用于表示本地向量。

```
case class LabeledPoint(label: Double, features: Vector) extends Product
with Serializable
```

LabeledPoint 中不仅包含一个 Vector 类型成员字段 features，还包含一个 Double 类型成员字段 label。features 字段表示一条数据的特征向量，label 字段则表示一条数据的目标值。MLlib 在处理监督式机器学习问题时，通常将训练集中每条数据表示为 LabeledPoint 类型的对象，而每条数据对应的 label 字段表示该数据的标识信息，是分类和回归分析过程中的重要依据。如果是二元分类问题，那么 label 字段可以取值 Double 型的 0.0 和 1.0；如果是多元分类问题，那么可以取值 0.0,1.0,2.0,…。

因此 LabeledPoint 在基于监督的学习算法中是一种非常常见的数据对象。LabeledPoint 类型的对象分为稀疏和密集两种。

示例 8-4

```
val pos = LabeledPoint(1.0, Vectors.dense(1.0, 0.0, 3.0))
val neg = LabeledPoint(0.0, Vectors.sparse(3, Array(0, 2), Array(1.0, 3.0)))
```

运行结果：

```
pos: org.apache.spark.mllib.regression.LabeledPoint = (1.0,[1.0,0.0,3.0])
neg: org.apache.spark.mllib.regression.LabeledPoint = (0.0,(3,[0,2],
[1.0,3.0]))
```

示例 8-4 分别创建了 pos 和 neg 两个 LabeledPoint 对象，其中 pos 封装了一个实数 1.0 表示数据的目标，还包含了一个密集向量表示数据的特征；neg 中的目标为 0.0，特征为一个稀疏向量。

8.2.2 MLlib 矩阵

MLlib 同时支持本地矩阵和分布式矩阵。

1. 创建本地矩阵

MLlib 的本地矩阵对应 Matrix 数据类型。本地矩阵具有整型的行、列索引值和实型的元素值，存储在本地节点。本地矩阵支持稀疏矩阵和密集矩阵。对于密集矩阵，可以以列优先（column-major order）的顺序存储在一维数组中。

示例 8-5

```
import org.apache.spark.mllib.linalg.{Matrix, Matrices}
val dm: Matrix = Matrices.dense(3, 2, Array(1.0, 3.0, 5.0, 2.0, 4.0, 6.0))
val sm: Matrix = Matrices.sparse(3, 2, Array(0, 1, 3), Array(0, 2, 1), Array(9,
6, 8))
```

运行结果：

```
dm: org.apache.spark.mllib.linalg.Matrix =
1.0  2.0
3.0  4.0
5.0  6.0
sm: org.apache.spark.mllib.linalg.Matrix =
3 x 2 CSCMatrix
(1,0) 9.0
(2,1) 6.0
(2,1) 8.0
```

示例 8-5 中 dm 被创建为密集矩阵((1.0, 2.0), (3.0, 4.0), (5.0, 6.0))；sm 被创建为稀疏矩阵 ((9.0,0.0),

(0.0, 8.0), (0.0, 6.0))。

2. 创建分布式矩阵

分布式矩阵具有长整型的行、列索引值和实型的元素值，数据存储在多个 RDD 中。由于转置一个分布式矩阵通常涉及全局的 shuffle 操作，所以分布式矩阵的存储方式非常重要。MLlib 中实现了三种不同的分布式矩阵：行矩阵（RowMatrix）、索引行矩阵（IndexedRowMatrix）、坐标矩阵（CoordinateMatrix）。

（1）行矩阵是一个面向行的分布式矩阵，其不具有行索引，也不具有列索引，其中每一行都是一个本地的向量。如果行矩阵的列数不是很多的话，那么本地存储的这些行，就可以在本地节点进行计算。

```
class RowMatrix extends DistributedMatrix with Logging
```

示例 8-6 Spark-shell 下创建行矩阵

```
import org.apache.spark.{SparkConf,SparkContext}
import org.apache.spark.mllib.linalg.{Vectors,Vector}
import org.apache.spark.mllib.linalg.distributed.RowMatrix
import org.apache.spark.rdd.RDD
val
rows:RDD[Vector]=sc.makeRDD(Seq(Vectors.dense(1,2,3),Vectors.dense(4,5,6)))
val mat: RowMatrix = new RowMatrix(rows)
val m = mat.numRows()
val n = mat.numCols()
println(m)
println(n)
```

运行结果：

```
rows:  org.apache.spark.rdd.RDD[org.apache.spark.mllib.linalg.Vector]  =
ParallelCollectionRDD[0] at makeRDD at <console>:28
mat: org.apache.spark.mllib.linalg.distributed.RowMatrix =
org.apache.spark.mlliblinalg.distributed.RowMatrix@50d8e2d5
m: Long = 2
n: Long = 3
```

上述代码首先创建 RDD[Vector]，得到 rows；并以 rows 作为参数直接创建行矩阵 mat；最后统计 mat 的行数与列数。

（2）索引行矩阵是对行矩阵的扩展，其中还包含每一行的行索引信息。一个索引行矩阵由包含 IndexedRow 类型的 RDD 构成，而 IndexedRow 中通过 index 成员字段封装了矩阵中一行的索引信息，vector 则给出了矩阵中一行的数据信息。

```
case class IndexedRow(index: Long, vector: Vector) extends Product with
Serializable
```

```
class IndexedRowMatrix (
    val rows: RDD[IndexedRow],
    private var nRows: Long,
    private var nCols: Int) extends DistributedMatrix
```

示例 8-7　在 Spark-shell 下创建索引行矩阵

```
import org.apache.spark.{SparkConf,SparkContext}
import org.apache.spark.mllib.linalg.{Vectors,Vector}
import org.apache.spark.mllib.linalg.distributed.RowMatrix
import org.apache.spark.rdd.RDD
import org.apache.spark.mllib.linalg.distributed.{IndexedRow,
IndexedRowMatrix,RowMatrix}
val iv1=new IndexedRow(0,Vectors.dense(1,2,3))
val iv2=new IndexedRow(1,Vectors.dense(4,5,6))
val rows: RDD[IndexedRow] = sc.makeRDD(Seq(iv1,iv2))
val mat: IndexedRowMatrix = new IndexedRowMatrix(rows)
val m = mat.numRows()
val n = mat.numCols()
println(m)
println(n)
```

运行结果：

```
iv1: org.apache.spark.mllib.linalg.distributed.IndexedRow =
IndexedRow(0,[1.0,2.0,3.0])
iv2: org.apache.spark.mllib.linalg.distributed.IndexedRow =
IndexedRow(1,[4.0,5.0,6.0])
mat: org.apache.spark.mllib.linalg.distributed.IndexedRowMatrix =
org.apache.spark.mllib.linalg.distributed.IndexedRowMatrix@5c00af3
m: Long = 2
n: Long = 3
```

示例 8-7 首先创建 IndexRow 类型对象 iv1、iv2；通过 makeRDD 将 iv1、iv2 封装成 RDD，进而得到 rows；再以 rows 作为参数构造 IndexedRowMatrix 类型对象 mat；最后统计 mat 的行数与列数。

（3）坐标矩阵由包含 MatrixEntry 类型的 RDD 构成，而 MatrixEntry 中封装了非零元素的行、列信息以及非零元素的值。

```
case class MatrixEntry(i: Long, j: Long, value: Double)
class CoordinateMatrix @Since("1.0.0") (
    @Since("1.0.0") val entries: RDD[MatrixEntry],
    private var nRows: Long,
    private var nCols: Long) extends DistributedMatrix
```

示例 8-8　在 Spark-shell 下创建坐标矩阵

```
import org.apache.spark.mllib.linalg.{Vectors,Vector}
```

```
    import org.apache.spark.mllib.linalg.distributed.RowMatrix
    import org.apache.spark.rdd.RDD
    import org.apache.spark.mllib.linalg.distributed.{CoordinateMatrix,
MatrixEntry}
    importorg.apache.spark.mllib.linalg.distributed.{IndexedRow,
IndexedRowMatrix,RowMatrix}
    val e1= new MatrixEntry(0,0,10)
    val e2= new MatrixEntry(1,1,20)
    val e3= new MatrixEntry(2,2,30)
    val e4= new MatrixEntry(3,3,40)
    val entries: RDD[MatrixEntry] = sc.makeRDD(Seq(e1,e2,e3,e4))
    val mat: CoordinateMatrix = new CoordinateMatrix(rdd)
    val m = mat.numRows()
    val n = mat.numCols()
    println(m)
    println(n)
```

运行结果：

```
    e1:org.apache.spark.mllib.linalg.distributed.MatrixEntry=MatrixEntry(0,0
,10.0)
    e2:org.apache.spark.mllib.linalg.distributed.MatrixEntry=MatrixEntry(1,1
,20.0)
    e3:org.apache.spark.mllib.linalg.distributed.MatrixEntry=MatrixEntry(2,2
,30.0)
    e4:org.apache.spark.mllib.linalg.distributed.MatrixEntry=MatrixEntry(3,3
,40.0)
    entries:org.apache.spark.rdd.RDD[org.apache.spark.mllib.linalg.distribut
ed.MatrixEntry]= ParallelCollectionRDD[0] at makeRDD at <console>:35
    mat:
org.apache.spark.mllib.linalg.distributed.CoordinateMatrix=org.apache.spark.
mllib.linalg.distributed.CoordinateMatrix@2c87ffab
    m: Long = 4
    n: Long = 4
```

示例 8-8 首先创建 MatrixEntry 类型对象 e1~e4；通过 makeRDD 将 e1~e4 封装成 RDD，进而得到 entries；再以 entries 作为参数构造 CoordinateMatrix 对象 mat；最后统计 mat 的行数与列数。

示例 8-9　在 Spark-shell 下基于 HDFS 中的数据创建坐标矩阵

```
    import org.apache.spark.mllib.linalg.{Vectors,Vector }
    import org.apache.spark.mllib.linalg.distributed.RowMatrix
    import org.apache.spark.rdd.RDD
    import org.apache.spark.mllib.linalg.distributed.{CoordinateMatrix,
MatrixEntry}
    import org.apache.spark.mllib.linalg.distributed.{IndexedRow,
```

```
IndexedRowMatrix,RowMatrix}
    val rdd=sc.textFile("hdfs://master:9000/user/dong/matrixdata.txt").
            map(line=>line.split(" ")).
    map(x=>{new MatrixEntry(x(0).toInt,x(1).toInt,x(2).toDouble)})
    val mat: CoordinateMatrix = new CoordinateMatrix(rdd)
    val m = mat.numRows()
    val n = mat.numCols()
    println(m)
    println(n)
```

matrixdata.txt 格式与内容如下：

```
0 0 1
1 1 2
2 2 3
2 1 3.1
3 0 4.1
3 3 5
```

运行结果：

```
    rdd:org.apache.spark.rdd.RDD[org.apache.spark.mllib.linalg.distributed.M
atrixEntry]=MapPartitionsRDD[30] at map at <console>:41
    mat:org.apache.spark.mllib.linalg.distributed.CoordinateMatrix=org.apach
e.spark.mllib.linalg.distributed.CoordinateMatrix@7fe0769d
    m: Long = 4
    n: Long = 4
```

示例 8-9 首先通过 textFile 逐行读取 hdfs://master:9000/user/dong/matrixdata.txt 中数据；然后以空格作为切分依据将每行切分出的数据封装成一个 MatrixEntry 类型的对象，进而得到 MatrixEntry 类型的 RDD，即 rdd；最后以 rdd 作为参数创建坐标矩阵 mat，并统计 mat 的行数与列数。

8.3 Spark MLlib 分类算法

分类属于监督式学习。在分类问题中，训练集包含的每一条数据都有特征和目标，其中目标是事先人为标注的。利用训练集与目标系统建立一个分类函数或分类模型，应用分类模型，能把数据库中未被分类的数据进行归类。Spark MLlib 目前支持的分类算法有逻辑回归、支持向量机、朴素贝叶斯和决策树。分类在机器学习和数据挖掘中是一项重要的任务，目前在商业上具有非常广泛的应用，典型的应用场景有流失预测、精准营销等。

8.3.1 贝叶斯分类算法

1. 贝叶斯定理简介

贝叶斯定理是描述随机事件 A 和 B 条件概率（或边缘概率）之间关系的一则定理。通常，在事件 B 发生条件下事件 A 发生的概率与在事件 A 发生条件下事件 B 发生的概率是不一样的，贝叶斯定理揭示了二者之间的联系。

P(A|B)表示事件B已经发生的前提下事件A发生的概率，叫做事件B发生下事件A的条件概率，其基本公式为：

$$P(A|B)= P(AB)/P(B)$$

类似的，P(B|A)表示事件 A 已经发生的前提下事件 B 发生的概率，其基本公式为：

$$P(B|A)= P(AB)/P(A)$$

贝叶斯定理描述了二者之间的关系：

$$P(B|A)= P(A|B)P(B)/P(A)$$

Spark MLlib 的贝叶斯分类模型采用朴素贝叶斯分类算法，即在贝叶斯定理的基础上假设数据的特征属性彼此独立。模型主要是计算每个类别的先验概率、各类别下各个特征属性的条件概率。在 LabeledPoint 类型训练集中，对具有同一目标的样本数据采用 combineByKey 聚合统计操作；再根据聚合结果计算先验概率、条件概率，得到朴素贝叶斯分类模型。在预测时，根据模型的先验概率、条件概率，计算每个样本属于每个类别的概率，最后取最大项作为样本的类别。

2. 模型的创建与使用

第 1 步，数据准备。

在 MLlib 中，朴素贝叶斯算法需要一个 LabeledPoint 类型的 RDD 作为训练集，对于 n 元分类问题，其中的 label 字段取值于{0,1, … , n-1}。

第 2 步，训练模型。

NaiveBayes 伴生对象中提供了用于训练模型的 train 方法，用户可以直接调用该方法创建朴素贝叶斯模型。train 方法通过设置训练参数进行模型训练，其主要参数如下。

input：训练集，LabeledPoint 类型的 RDD。

lambda：平滑参数。

第 3 步，使用模型。

NaiveBayes 中 train 方法的返回值为 NaiveBayesModel 类型，其中定义了对特征的目标值进行预测的 predict 方法。该方法既可以对单条特征（一个向量）进行预测，也可以将若干条特征封装成 RDD[Vector]交由 predict 进行批量预测。

需要注意，负数不能作为特征。

3. 案例

示例 8-10　在 IntelliJ IDEA 中创建并使用朴素贝叶斯模型

创建训练集数据 nb_traindata.txt ，并存放在/home/dong/mllibdata 目录下。

```
0,1 0 0
0,2 0 0
0,4 0 0
1,0 2 0
1,0 3 0
1,0 4 0
2,0 0 1
2,0 0 2
2,0 0 3
```

创建测试集数据 nb_testdata.txt，并存放在/home/dong/mllibdata 目录下，如图 8.3 所示。

```
3 0 0
0 1 0
0 0 4
```

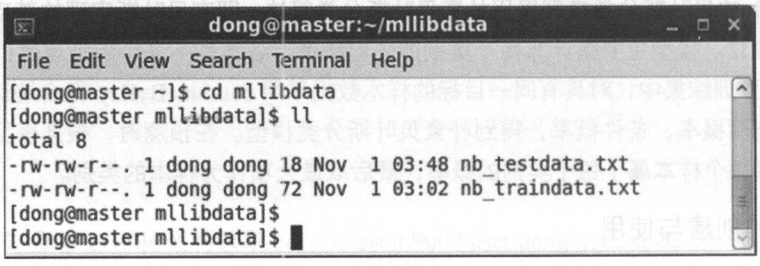


图 8.3　测试数据集目录示意图

启动 IntelliJ IDEA，编辑如下的 Spark MLlib 应用程序：

```
import org.apache.log4j.{Level, Logger }
import org.apache.spark.{SparkConf, SparkContext}
import org.apache.spark.mllib.classification.{NaiveBayes,
NaiveBayesModel}
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.regression.LabeledPoint
object NaiveBayesTest{
    def main(args:Array[String]){
//构建 SparkContext 类型对象，封装应用程序属性信息与环境信息。
        val conf = new SparkConf().setAppName("NaiveBayesTest").
                setMaster("spark://192.168.149.132:7077")
        val sc = new SparkContext(conf)
//读取并创建训练数据。
        val data=sc.textFile("/home/dong/mllibdata/nb_traindata.txt")
```

```
        val trainData=data.map{line=>
                val parts=line.split(",")
                LabeledPoint(parts(0).toDouble,Vectors.dense(parts(1).sp
            lit("").
                map(_.toDouble)))
        }
//读取测试数据。
        val testData=sc.textFile("/home/dong/mllibdata/nb_testdata.txt").

                        map{line=>val parts=line.split(" ")
                                Vectors.dense(parts.map(_.toDouble))
                            }
//训练模型。
        val model = NaiveBayes.train(trainData,lambda=1.0)
//使用模型对单条特征进行预测。
        println("predict:" + model.predict(Vectors.dense("0,0,3".split(",").
map(_.toDouble))))
//使用模型对批量特征进行预测,并将结果保存至文件夹 naivebyesresult 中。
        val result=testData.map{line=>
            {   val prediction=model.predict(line)
                line+" belongs to class "+prediction
            }
        }.saveAsTextFile("/home/dong/naivebyesresult")
    }
}
```

运行结果：

IntelliJ IDEA 中运行结果如图 8.4 所示，本地目录下生成的预测结果数据集如图 8.5 和图 8.6 所示。

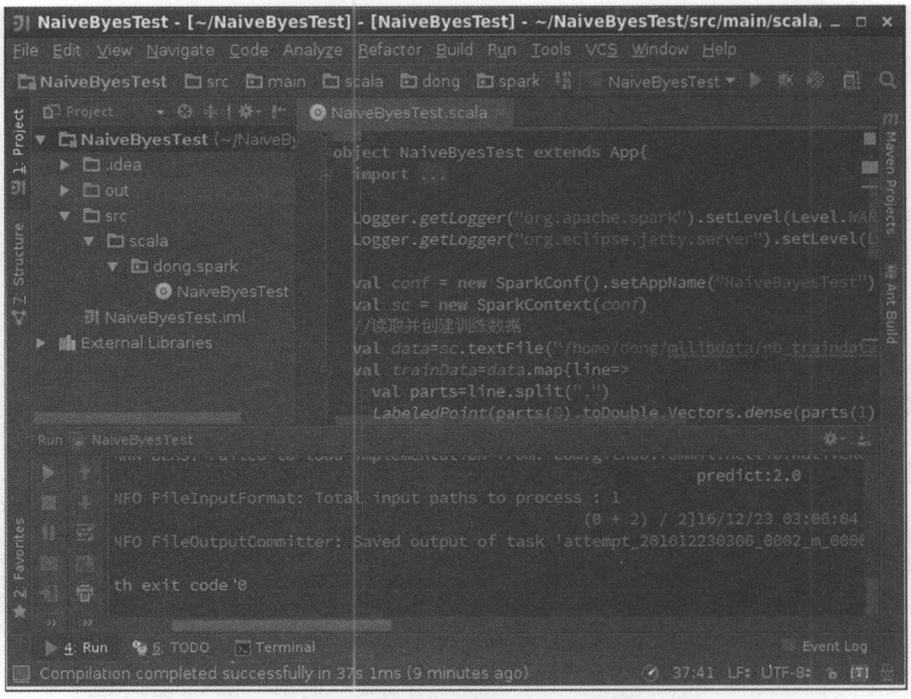


图 8.4　NaiveBayesTest 运行结果图

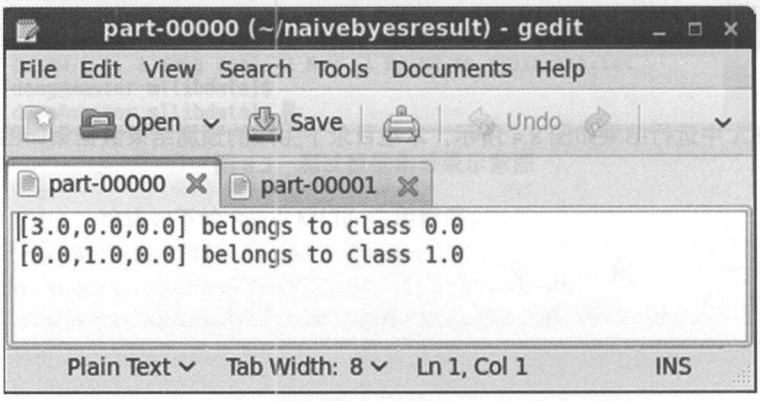


图 8.5　本地目录下生成的预测结果数据集 part-00000

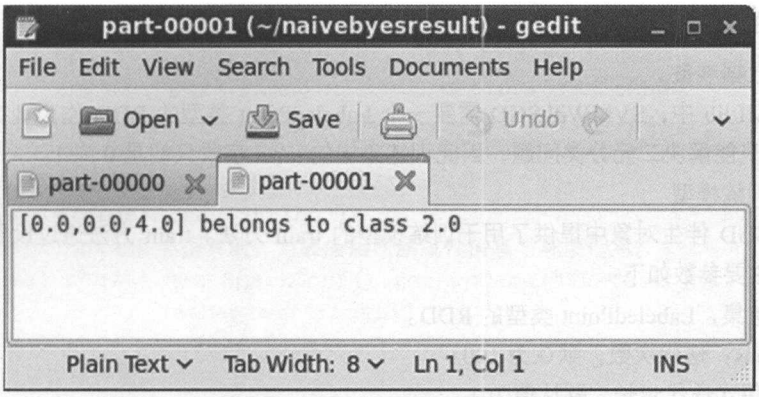


图 8.6 本地目录下生成的预测结果数据集 part-00001

8.3.2 支持向量机算法

1. 支持向量机简介

支持向量机（Support Vector Machine, SVM）是一种分类算法，通过寻求结构化风险最小来提高学习机的泛化能力，实现经验风险和置信范围的最小化，从而达到在统计样本量较少的情况下，也能获得良好的统计结果。SVM 通常用于解决监督式机器学习中的二元分类问题，其基本模型定义为特征空间上间隔最大的线性分类器，即支持向量机的学习策略便是间隔（Gap）最大化，最终可转化为一个凸二次规划问题的求解，如图 8.7 所示。

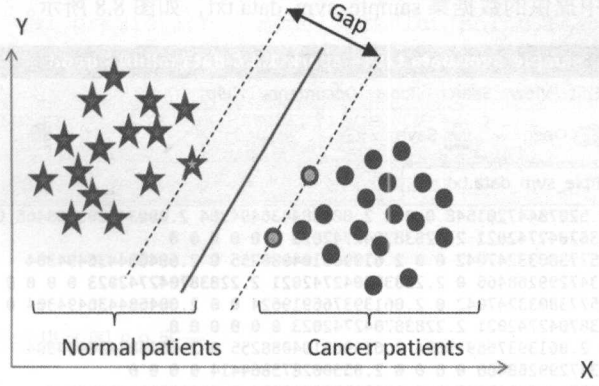


图 8.7 SVM 最优分割超平面

Spark MLlib 通过 SVMWithSGD 类和其伴生对象实现了线性 SVM 二元分类模型，并采用随机梯度下降算法来优化目标函数。

2. 模型的创建与使用

第 1 步，数据准备。

在 Spark MLlib 中，SVMWithSGD 需要一个 LabeledPoint 类型的 RDD 作为训练集。但是由于 SVMWithSGD 只能解决二元分类问题，因此训练集中的 label 字段只能是 0 或 1。

第 2 步，训练模型。

SVMWithSGD 伴生对象中提供了用于训练模型的 train 方法。train 方法通过设置训练参数进行模型训练，其主要参数如下：

input：训练集，LabeledPoint 类型的 RDD。

numIterations：迭代次数，默认为 100。

stepSize：每次迭代步长，默认值为 1。

miniBatchFraction：每次迭代参与计算的样本比例，默认值为 1.0，表示全部样本参与计算。

initialWeights：初始化权重。

regParam：正则因子，默认采用 L2 正则化。

第 3 步，使用模型。

SVMWithSGD 中的 train 方法返回值为 SVMModel 类型，其中定义了对特征的目标值进行预测的 predict 方法。该方法既可以对单条特征（一个向量）进行预测，也可以将若干条特征封装成 RDD[Vector]交由 predict 进行批量预测。

3. 案例

示例 8-11　在 IntelliJ IDEA 中创建并使用支持向量机模型

使用 spark-1.4.0 中提供的数据集 sample_svm_data.txt，如图 8.8 所示。

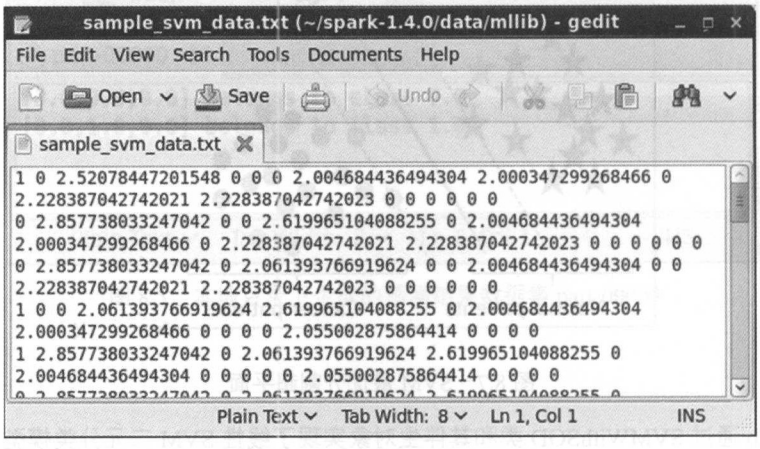


图 8.8　数据集 sample_svm_data.txt 示意图

启动 IntelliJ IDEA，编辑如下的 Spark MLlib 应用程序：

```
import org.apache.log4j.{Level, Logger }
import org.apache.spark.{SparkConf, SparkContext}
import org.apache.spark.mllib.classification.SVMWithSGD
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.linalg.{Vectors,Vector }
object SVMTest{
    def main(args:Array[String]){
//构建 SparkContext 类型对象，封装应用程序属性信息与环境信息。
        val conf = new SparkConf().setAppName("SVMTest").
                   setMaster["local[2]"]
        val sc = new SparkContext(conf)
//读取并创建训练数据。
         val data =
    sc.textFile("/home/dong/spark-1.4.0/data/mllib/
    sample_svm_data.txt")
        val parsedData = data.map { line =>
        val parts = line.split(' ')
        LabeledPoint(parts(0).toDouble,Vectors.dense(parts.tail.map
    (x=>x.toDouble).toArray))
    }
 //设置迭代次数并进行训练。
    val numIterations = 20
//训练模型。
    val model = SVMWithSGD.train(parsedData, numIterations)
//预测并统计分类错误的样本比例。
    val labelAndPreds = parsedData.map { point =>
              val prediction = model.predict(point.features)
              (point.label, prediction)
       }
       val trainErr = labelAndPreds.filter(r => r._1 !=r._2).
                  count.toDouble/parsedData.count
       println("Training Error = " + trainErr)
    }
}
```

运行结果：

IntelliJ IDEA 中运行结果如图 8.9 所示。

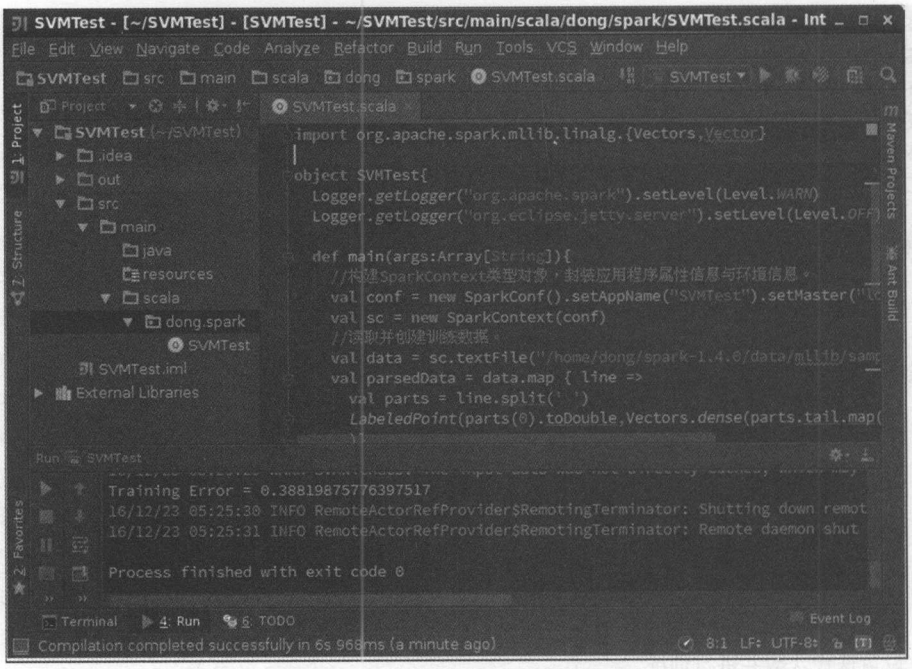


图 8.9　SVMTest 运行结果图

8.3.3　决策树算法

1．决策树简介

决策树（Decision Tree, DT）是一个类似于流程图的树结构，其中每个内部节点表示在一个特征属性上的测试。树的最顶层节点是根节点，每个分枝代表一个测试输出，而每个树叶节点代表类或者类分布。用决策树进行决策的过程就是从根节点开始，测试分类项中相应的特征属性，并按照其值选择输出分支，直至到达叶节点，并将叶节点存放的类别作为决策结果。图 8.10 是一棵预测已购买保险的客户是否续保的决策树，其中房产、婚姻、年收入情况表示客户的特征属性，续保和不续保表示类别。

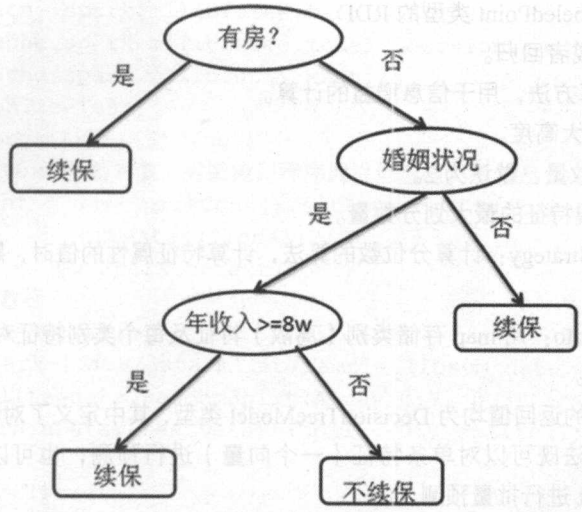


图 8.10 客户续保决策示意图

决策树算法的核心问题是选取特征属性，每次争取能够选取出最有助于分类的属性（例如图 8.10 中为何最先选择客户的房产属性作为决策树根节点）。决策树的算法原理决定了其适合处理数据特征属性较少的分类问题。在构建决策树的过程中，当前选择哪个特征属性（也叫分裂规则）取决于切分后节点数据集中类别（分区）的有序程度（纯度）。划分后的分区数据越纯，那么当前分裂规则就越合适。衡量节点数据集合的纯度的指标有：熵、基尼和方差，其中熵和基尼是针对分类的，方差是针对回归的。

Spark MLlib 决策树模型的训练过程就是决策树的构建过程，它采用自顶向下的递归方式，在决策树的内部节点进行特征属性的比较，并根据不同的特征属性值选择该节点下的分支，重复这一判断选择过程直到满足一定的终止条件为止。Spark MLlib 通过 DecisionTree 类和其伴生对象实现了决策树模型。由于该模型同时支持连续型和离散型的目标值，因此不仅可用于解决二元、多元分类问题，还适用于机器学习中的回归问题。

2. 模型的创建与使用

第 1 步，数据准备。

在 MLlib 中，DecisionTree 需要一个 LabeledPoint 类型的 RDD 作为训练集。对于 n 元分类问题，训练集中的 label 字段的值取于 $\{0,1,\ldots,n\text{-}1\}$；对于回归问题，训练集中的 label 字段的值为任意实数。

第 2 步，训练模型。

DecisionTree 伴生对象中提供了用于训练模型的 train 方法。由于决策树模型同时适用于机器学习中的回归和分类问题，因此其训练模型的方法又分为三种：train（训练回归和分类模型）、trainClassifier（训练分类决策树模型）、trainRegressor（训练回归决策树模型）。train 方法通过设置训练参数进行模型训练，其主要参数如下：

input：训练集，LabeledPoint 类型的 RDD。

algo：算法，分类或者回归。

impurity：纯度计算方法，用于信息增益的计算。

maxDepth：树的最大高度。

numClasses：分类数量，默认为 2。

maxBins：用于分裂特征的最大划分数量。

quantileCalculationStrategy：计算分位数的算法，计算特征属性的值时，默认使用 sort 策略，即将属性值排序。

categoricalFeaturesInfo：用 map 存储类别（离散）特征及每个类别特征对应值（类别）的数量。

第 3 步，使用模型。

上述训练模型方法的返回值均为 DecisionTreeModel 类型，其中定义了对特征的目标值进行预测的 predict 方法。该方法既可以对单条特征（一个向量）进行预测，也可以将若干条特征封装成 RDD[Vector]交由 predict 进行批量预测。

3．案例

示例 8-12　在 IntelliJ IDEA 中创建并使用决策树模型

采用 Spark-1.4.0 提供的数据集 sample_libsvm_data.txt，如图 8.11 所示。

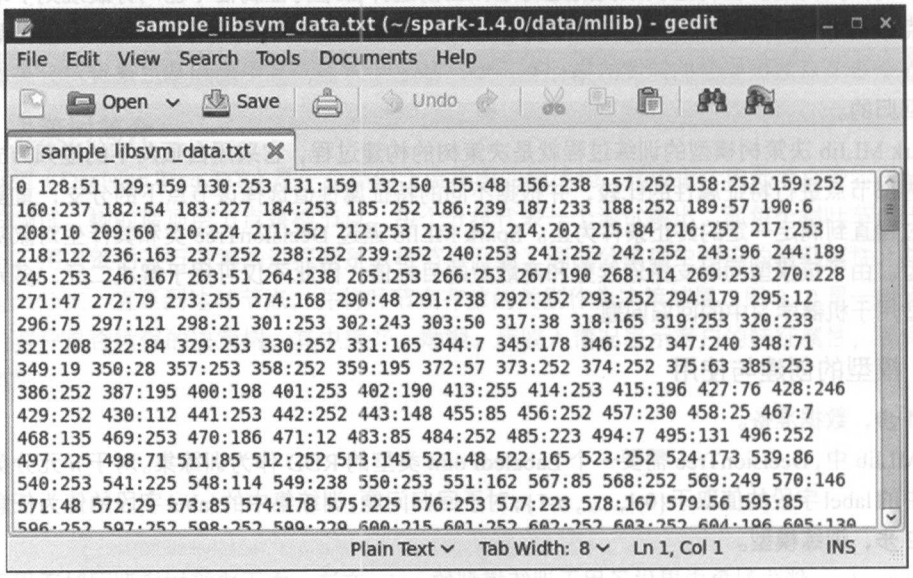


图 8.11　数据集 sample_libsvm_data.txt 示意图

启动 IntelliJ IDEA，编辑如下的 Spark MLlib 应用程序：

```
import org.apache.log4j.{Level, Logger }
import org.apache.spark.{SparkConf, SparkContext}
```

```
import org.apache.spark.mllib.tree.DecisionTree
import org.apache.spark.mllib.tree.model.DecisionTreeModel
import org.apache.spark.mllib.util.MLUtils
object DecisionTreeTest{
    def main(args:Array[String]){
//构建 SparkContext 类型对象，封装应用程序属性信息与环境信息。
        val conf = new SparkConf().setAppName("DecisionTreeTest").
                setMaster("local[2]")
        val sc = new SparkContext(conf)
//读取并创建训练数据。
        val data=MLUtils.loadLibSVMFile(sc,
"/home/dong/spark-1.4.0/data/mllib/sample_libsvm_data.txt")
        val splits=data.randomSplit(Array(0.7, 0.3))
        val (trainData, testData)=(splits(0), splits(1))
//训练模型。
        val numClasses=2
        val categoricalFeaturesInfo=Map[Int, Int]()
        val impurity ="gini"
        val maxDepth=5
        val maxBins=32
        valmodel=DecisionTree.trainClassifier(trainData,numClasses,
                categoricalFeaturesInfo,impurity,maxDepth,maxBins)
//使用模型对批量特征进行预测。
        val labeleIAndPreds = testData.map{ point =>
                val prediction = model.predict(point.features)
                (point.label, prediction)
            }
        val testErr=labeIAndPreds.filter(r=>r._1!=r._2).
                                count.toDouble/
                                testData.count()
        println("Test Error = " + testErr)
        println("Learned classification tree model:\n" + model.
toDebugString)
    }
}
```

运行结果：

在 IntelliJ IDEA 中运行结果如图 8.12 所示。

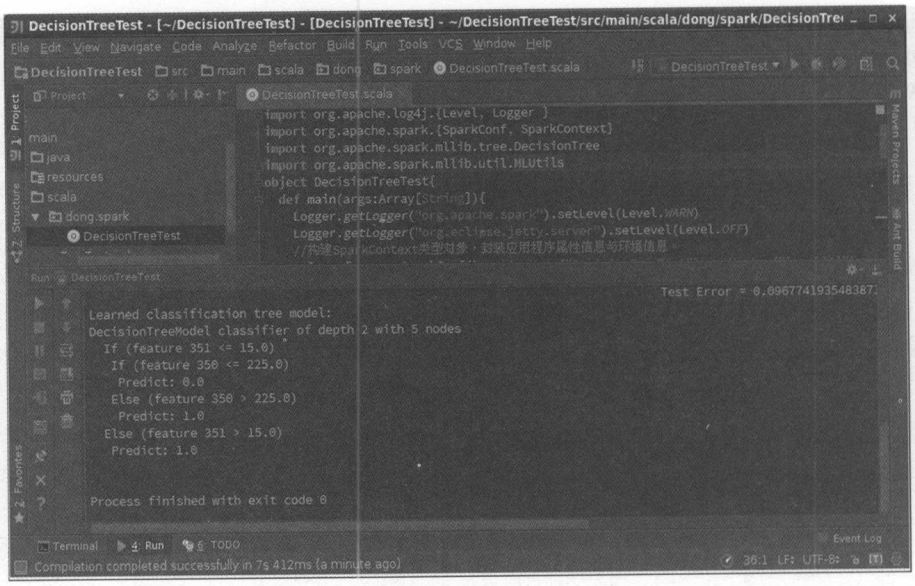


图 8.12　DecisionTreeTest 运行结果图

8.4 MLlib 线性回归算法

1．线性回归简介

线性回归（Linear Regression）是利用线性回归方程（函数）对一个或多个自变量和因变量之间关系进行建模的一种回归分析方法。线性回归问题属于监督学习范畴，要求训练数据集中的数据具有明确的目标。

若回归分析中只包含一个自变量和一个因变量，且二者的关系可用一条直线近似表示，那么这种回归分析被称为一元线性回归分析，或一元线性拟合，如图 8.13 所示。如果回归分析中包含两个或两个以上的自变量，且因变量和自变量之间是线性关系，则称为多元线性回归分析。通常初学者对于线性回归较难理解，若换个角度来解释，其实线性回归问题就是线性拟合问题。以一元线性回归为例，就是研究如何确定一条直线来近似地表示描述训练集中所有数据的空间分布。类似地，对于二元回归问题，采用一个平面来拟合；若多元回归问题，采用超平面来拟合。线性拟合中用于拟合的直线、平面、超平面都对应着线性方程，因此线性回归问题转换为求解相应线性方程（预测函数）的问题。用户可以通过求解出的线性方程对数据的目标值进行预测。

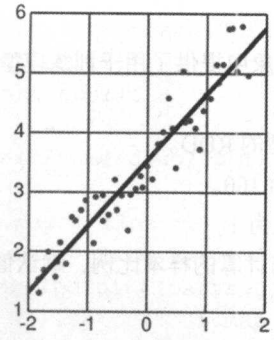


图 8.13 一元线性拟合示意图

例如二元线性回归方程可以描述如下：

$$h(x) = h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2$$

其中 $h(x)$ 为预测函数，$\theta_i (i=1,2,..,n)$称为估计参数，模型训练的目的就是计算出这些参数的值。

类似的，无论是一元还是多元线性回归方程可统一表示为如下形式：

$$h_\theta(x) = \theta^T X$$

因此确定预测函数的过程转换为求解θ^T的过程。预测函数可以有多个，例如图 8.13，还可以画出多条拟合直线，我们要确定出最优的预测函数，即对于求解出的多个θ^T，需要一个机制去评估和选择θ^T。为此引入损失函数（loss function）$J(\theta)$用于描述一条样本数据采用函数 $h(x)$得到的预测目标值与其真实目标值之间的偏差程度。

$$J(\theta) = \frac{1}{2}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2$$

$$\min_\theta J_\theta$$

显然，$J(\theta)$函数的值越小表示预测函数 $h(x)$越准确，因此需要找到 $J(\theta)$函数的最小值。求解一个函数最小值的方法有多种，Spark MLlib 采用的是随机梯度下降算法：在每次迭代中，随机抽取一定比例的样本数据进行当前的迭代计算；对计算样本集中的每一个样本分别计算梯度（分布式计算每个样本的梯度）；然后再通过聚合函数对样本的梯度进行累加，得到该样本的平均梯度及损失；最后根据最新的梯度及上次迭代的权重进行权重的更新。Spark MLlib 通过 LinearRegressionWithSGD 类和其伴生对象实现了基于随机梯度下降的线性回归模型。

2. 模型的创建与使用

第 1 步，数据准备。

在 MLlib 中，LinearRegressionWithSGD 需要一个 LabeledPoint 类型的 RDD 作为训练集。训练集中 label 字段的值可以是任意实数。

第 2 步，训练模型。

LinearRegressionWithSGD 伴生对象中提供了用于训练模型的 train 方法。train 方法通过设置训练参数进行模型训练，其主要参数如下：

input：训练集，LabeledPoint 类型的 RDD。

numIterations：迭代次数，默认为 100。

stepSize：每次迭代步长，默认值为 1。

miniBatchFraction：每次迭代参与计算的样本比例，默认值为 1.0，表示全部样本参与计算。

initialWeights：初始化权重。

第 3 步，使用模型。

LinearRegressionWithSGD 中 train 方法的返回值为 LinearRegressionModel 类型，其中定义了对特征的目标值进行预测的 predict 方法。

3. 案例

示例 8-13　在 IntelliJ IDEA 中创建并使用线性回归模型

采用 Spark-1.4.0 提供的数据集 ridge-data.txt，如图 8.14 所示。

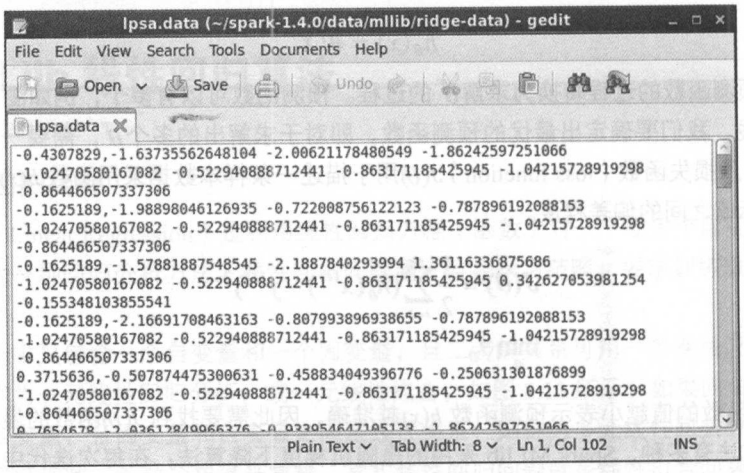


图 8.14　数据集 ridge-data.txt 示意图

启动 IntelliJ IDEA，编辑如下的 Spark MLlib 应用程序：

```
import org.apache.log4j.{Level, Logger }
import org.apache.spark.{SparkConf, SparkContext}
import org.apache.spark.mllib.regression.LinearRegressionWithSGD
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.linalg.{Vectors,Vector }
object LinearRegressionTest extends App{
//构建 SparkContext 类型对象，封装应用程序属性信息与环境信息
```

```
    val conf=new SparkConf().setAppName("LinearRegressionTest").
             setMaster("local[2]")
    val sc= new SparkContext(conf)
//读取并创建训练数据。
    val data = sc.textFile("/home/dong/spark-1.4.0/data/mllib/ridge-data/
lpsa.data")
    val parsedData = data.map { line =>
                  val parts = line.split(",")
                      LabeledPoint(parts(0).toDouble,Vectors.dense(part
                      s (1).split(" ").map (x => x.toDouble)))
                  }
//训练模型。
    val numIterations = 100
    val model = LinearRegressionWithSGD.train(parsedData, numIterations)
//预测并统计回归错误的样本比例。
    val valuesAndPreds = parsedData.map { point =>
                  val prediction = model.predict(point.features)
                  (point.label, prediction)
                  }
    val MSE = valuesAndPreds.map{ case(v, p) => math.pow((v - p), 2)}.reduce(_
+ _)/valuesAndPreds.count
    println("training Mean Squared Error = " + MSE)
}
```

运行结果：

在 IntelliJ IDEA 中运行结果如图 8.15 所示。

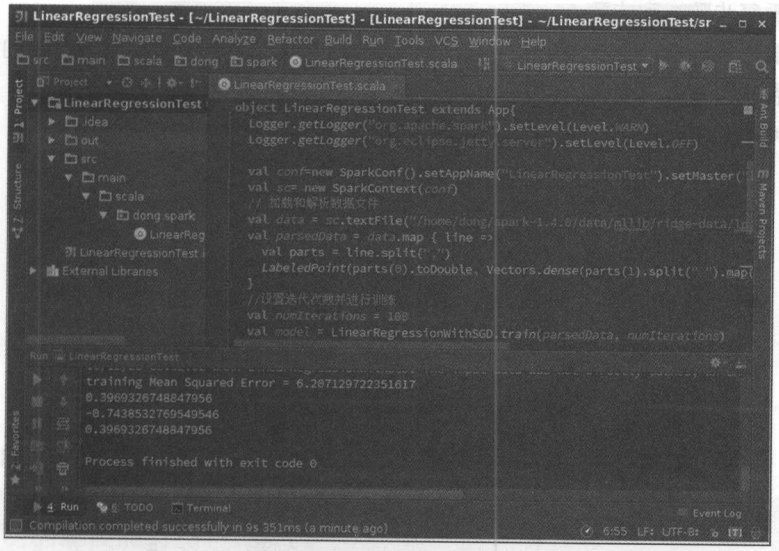


图 8.15 LinearRegression 运行结果图

8.5 MLlib 聚类算法

1. K-Means 聚类简介

聚类是观察式学习，在聚类前可能不明确类别甚至不知道类别的数量，因此聚类属于无监督学习的一种。

聚类算法通常被用于处理探索性的分析问题，是根据“物以类聚”的原理，将本身没有类别的样本聚集成不同的组，每一组称为一簇。聚类的目的是使得属于同一簇内的样本之间尽可能的相似，而不同簇的样本之间差异最大化。常见的典型应用场景有客户细分、市场细分、价值评估等。MLlib 目前支持被广泛使用的 K-Means 聚类算法。本节重点介绍 K-Means 算法。

K-Means 方法是处理聚类问题的一种最常用方法，属于基于平方误差的迭代重分配聚类算法，其核心步骤如下。

第 1 步，选定 K 个中心点。

中心点初值的选取对聚类结果影响很大，Spark MLlib 中可以采用随机选取和 k-Means++的方法。

第 2 步，构建簇。

对训练集中的每个样本点，计算该样本点分别到 K 个中心点的距离，并将其指配给离它最近的那个中心点。经过步骤 2，指派给同一中心点的所有样本点形成一个簇，K 个中心对应形成 K 个簇。

第 3 步，重新计算每个簇的中心点。

每个簇的中心点即簇的质心，可简单地采用簇内各维坐标的算术平均进行计算。需要注意，重新计算出的中心点未必是训练集中的样本点。

第 4 步，重复步骤 2 和步骤 3。

在重复步骤 2 和步骤 3 的过程中，会有一些点被划入新的簇，一些点被划出旧的簇。重复这一过程直至中心点的位置不再变化，或变化小于一定阈值，或者达到迭代次数。

在图 8.16 中，K-Means 算法设定中心点个数为 2。

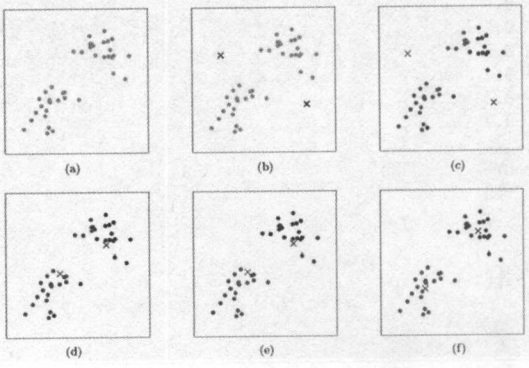


图 8.16　含有 2 个中心点的 K-Means 算法示意图

Spark MLlib 中定义了 KMeans 类和其伴生对象用来实现 K-Means 聚类方法。

2. 模型的创建与使用

第 1 步，数据准备。

由于 K-Means 方法用于处理无监督学习问题，训练集中的样本无须给出明确的目标。因此在 MLlib 中 K-Means 需要一个 Vector 类型的 RDD 作为训练集。

第 2 步，训练模型。

K-Means 伴生对象中提供了用于训练模型的 train 方法。train 方法通过设置训练参数进行模型训练，其主要参数如下：

data：训练集，Vector 类型的 RDD。

k：聚类数量。

maxIterations：最大迭代次数。

initializationMode：初始化中心，支持随机或者 k-Means++（默认）。

seed：初始化时的随机种子。

第 3 步，使用模型。

在 K-Means 中 train 方法的返回值为 K-Means Model 类型，其中定义了用于对特征进行预测的 predict 方法。

3. 案例

示例 8-14　在 IntelliJ IDEA 中创建并使用 K-Means 模型

采用 Spark-1.4.0 提供的数据集 kmeans-data.txt，如图 8.17 所示。

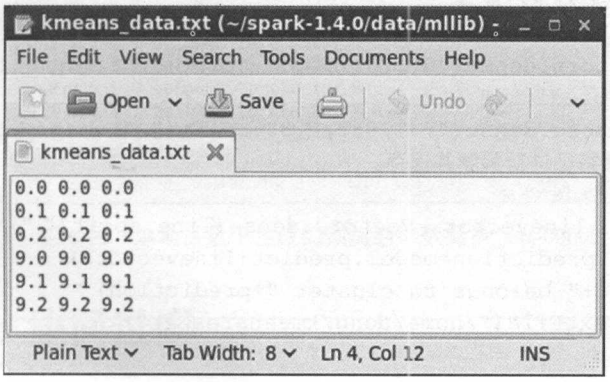


图 8.17　数据集 kmeans-data.txt 示意图

启动 IntelliJ IDEA，编辑如下的 Spark MLlib 应用程序：

```
import org.apache.log4j.Logger
import org.apache.log4j.Level
import org.apache.spark.{SparkContext, SparkConf}
```

```
import org.apache.spark.mllib.clustering.{KMeans,KMeansModel}
import org.apache.spark.mllib.linalg.{Vectors,Vector }
import org.apache.spark.mllib.linalg.distributed.RowMatrix
import org.apache.spark.rdd.RDD
object KMeansTest extends App{
    Logger.getLogger("org.apache.spark").setLevel(Level.WARN)
    Logger.getLogger("org.eclipse.jetty.server").setLevel(Level.OFF)
//构建 SparkContext 类型对象，封装应用程序属性信息与环境信息。
    val conf = new SparkConf().setAppName("Kmeans").setMaster("local[2]")
    val sc =new SparkContext(conf)
//读取并创建训练数据。
    val data=sc.textFile("/home/dong/spark-1.4.0/data/mllib/
kmeans_data.txt")
    val parseData=data.map(s=>Vectors.dense(s.split(" ").
                map(_.toDouble)))
//训练模型。
    val numClusters=2
    val numIterations=15
    val model=KMeans.train(parseData, numClusters, numIterations)
    println("Cluster centers")
    for(c<-model.clusterCenters){
       println(" " + c.toString)
    }
//评估模型。
    val cost = model.computeCost(parseData)
    println("Within the sum of squared Eerror : " + cost)
//使用模型对单条特征进行预测。
    println("predict:"                                                +
model.predict(Vectors.dense("0.1,0.2,0.3".split(",").map(_.toDouble))))
    println("predict:"                                                +
model.predict(Vectors.dense("7.1,8.2,9.3".split(",").map(_.toDouble))))
//使用模型对批量特征进行预测并保存。
    data.map{line=>
          {val linevectors=Vectors.dense(line.split(" ").map(_.toDouble))
          val prediction=model.predict(linevectors)
          line+" belongs to cluster "+prediction}
    }.saveAsTextFile("/home/dong/kmeansresult")
}
```

运行结果：

在 IntelliJ IDEA 中运行结果如图 8.18 所示，本地目录下生成的预测结果数据集如图 8.19 所示。

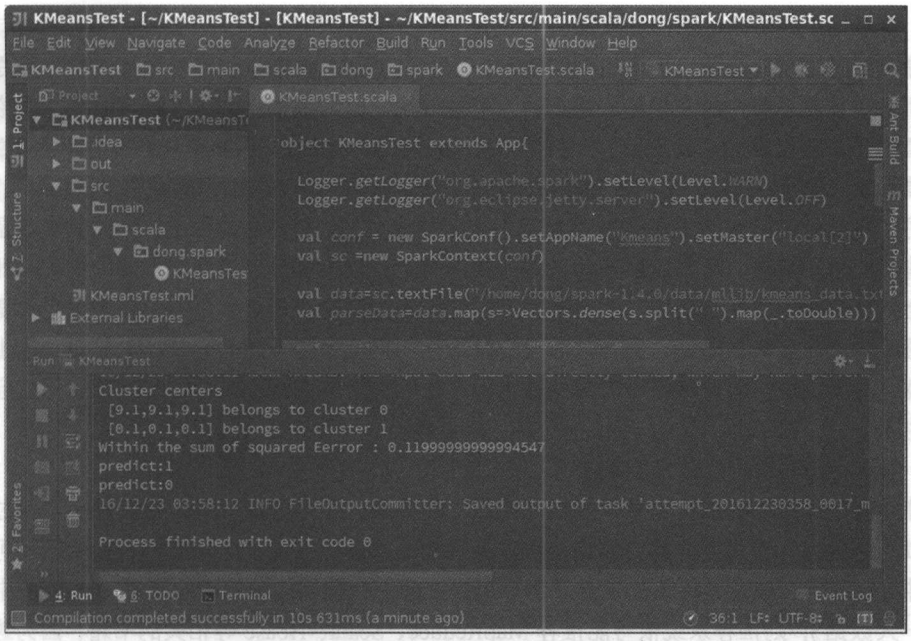


图 8.18 KMeansTest 运行结果图

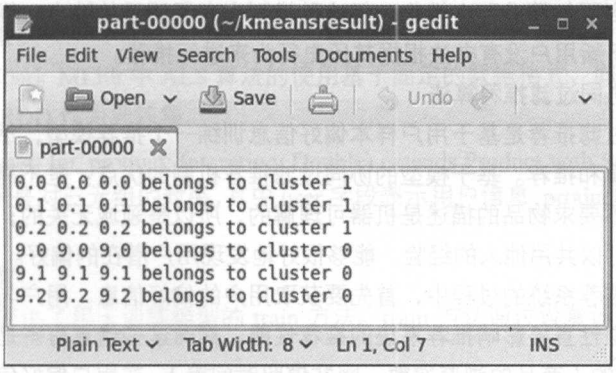


图 8.19 本地目录下生成的预测结果数据集 part-00000

8.6 MLlib 协同过滤

1. 协同过滤简介

协同过滤推荐算法是一种基于群体用户或者物品的典型推荐算法，也是目前推荐算法中最常用、最经典的算法。基于协同过滤算法的推荐系统在电子商务领域中有着非常广泛的应用。协同过滤推荐算法可以细分为基于用户的、基于物品的和基于模型的三类。

（1）基于用户的协同过滤推荐算法

这种模型通过用户的属性信息（例如年龄、职业、性别等社会学信息）来挖掘相似用户。当确定了某个（或某类）用户的“相似”用户后，可将该用户已经消费过的物品推荐给其“相似”的用户。例如一个 25 岁的女士 A 消费过一件风衣，对于另外一个 25 岁的女士 B，基于年龄和性别属性信息，推荐系统可认定 B 为 A 的一个相似用户。那么推荐系统有理由相信 A 与 B 对物品的偏好很可能相同，因此可以把 A 消费的风衣推荐给 B。这种模型不依赖物品的数据，因此其优点是：不依赖用户消费的历史数据，可以应用在新进入的用户身上；该模型可以适用于不同的物品，即便这些物品完全属于不同的类型或领域。但这种模型的缺点也很明显：相似用户的划分依据是关键，但这些依据通常是不易获取的隐私；另一方面没有考虑到用户偏好的不同，偏好是基于用户人生经历、生活环境、消费品位、消费水平等方面的综合而产生的，这些方面绝大部分是不能仅用社会学信息来概括的。

（2）基于物品的协同过滤推荐算法

这种模型通过物品的属性信息来挖掘“相似”的物品，在用户消费时，将其已经消费过的物品的“相似”物品推荐给用户。其中属性是代表某种物品特征的描述，比如某种书籍的属性可能就包括书籍种类、作者、技术领域等的信息。例如用户在图书销售网站购买或搜索过 Spark 的书籍（属性为 IT、大数据），则推荐系统很可能推荐 MapReduce、Scala、Hbase 等相关的书籍，因为这些都属于与 Spark“相似”的书籍。这种模型使用了用户消费的历史数据，优点是可以根据用户的已知消费行为来进行推荐，做到更加精准和个性化。但这种模型也有很明显的缺点：推荐质量受商品属性完整性的影响较大；对于新用户没有办法根据其历史行为来进行推荐。

（3）基于模型的协同过滤推荐算法

基于模型的协同过滤推荐是基于用户样本偏好信息训练一个推荐模型，然后根据实时的用户偏好信息进行预测、计算和推荐。基于模型的协同过滤推荐机制的优点：是不需要对物品或者用户进行严格的建模，而且不要求物品的描述是机器可理解的，所以是领域无关的；这种方法计算出来的推荐结果是开放的，可以共用他人的经验，能够很好地发现用户潜在的偏好。

在构建协同过滤推荐系统的过程中，首先要获取用户的偏好信息、用户与物品的属性信息，这些信息的有效性、完整性直接影响推荐系统的推荐性能，然而这些信息有些是显性的（购买历史、评分），有些则是隐形的（商品的浏览次数、网站停留时间等）。在用户偏好信息与物品属性信息的基础上，利用相似度度量方法（欧几里得距离、皮尔逊系数、Jaccard 系数、余弦角度计算等）确定相似用户或相似物品，并在此基础上再进行推荐计算。

Spark MLlib 基于交替最小二乘法（Alternating Least Squares, ALS）实现了基于模型的协同过滤推荐算法。该算法基于一个用户物品评分矩阵 R，该矩阵可以被分解为两个矩阵 U、V 乘积的形式。

$$U \times V = R$$

其中 U 表示用户矩阵、V 表示物品的矩阵。在 Spark MLlib 的 ALS 算法中，首先对 U 或者 V 矩阵随机化生成，之后固定某一个特定矩阵，去求解另外一个未随机化的矩阵；然后利用被求出的矩阵去求解随机化矩阵对象。两个矩阵相互迭代计算，直至求取出的矩阵与实际矩阵差异达到设定的

阈值范围。

例如图 8.20 是某电影网站基于 ALS 实现的推荐系统示意图，其中 Ratings 是所有用户对所有电影的评分矩阵、Users 为用户矩阵、Moives 为物品矩阵。通常 Ratings 是一个非常稀疏、数据量非常大的矩阵，其中的非空元素表示用户对已观看电影的评分。通过 ALS 算法预测用户对未观看电影的评分，进而将 Ratings 中的空元素补全，然后按照预测评分排序，截取评分最高的前几部电影推荐给用户。

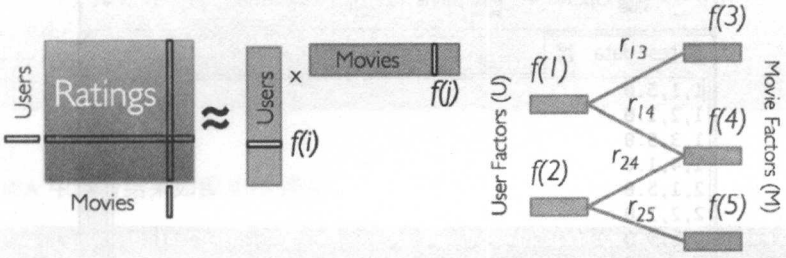


图 8.20 基于 ALS 的推荐系统示意图

Spark MLlib 中定义了 ALS 类和其伴生对象用来实现基于交替最小二乘法的协同过滤算法。

2. 模型的创建与使用

第 1 步，数据准备。

这里需要注意，Spark MLlib 中 ALS 算法的使用基于固定的数据格式，即 Rating。因此 ALS 需要一个 Rating 类型的 RDD 作为训练集。

```
case class Rating(user: Int, product: Int, rating: Double) extends Product with Serializable
```

Rating 类型的实例是对三元组的封装，其中 user 字段表示用户信息、product 字段表示物品信息、ratings 表示评分信息。

第 2 步，训练模型。

ALS 伴生对象中提供了用于训练模型的 train 方法。train 方法通过设置训练参数进行模型训练，其参数主要包括如下。

numBlocks：并行计算的 block 数（-1 为自动配置）。

rank：模型中隐藏因子数。

iterations：算法迭代次数。

lambda：ALS 中的正则化参数。

implicitPref：使用显示反馈 ALS 变量或隐式反馈。

alpha：ALS 隐式反馈变化率用于控制每次拟合修正的幅度。

第 3 步，使用模型。

ALS 中 train 方法的返回值为 ALSModel 类型，其中定义了用于对特征进行预测的 predict 方法。

3. 案例

示例 8-15　在 IntelliJ IDEA 中创建并使用 ALS 模型

采用 Spark-1.4.0 提供的数据集 test.data，如图 8.21 所示。

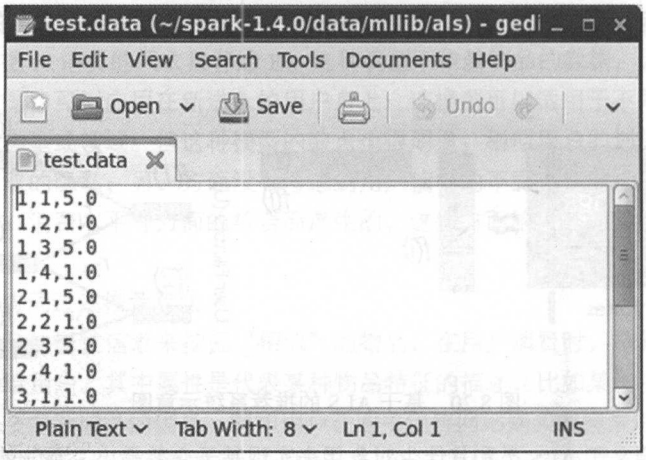


图 8.21　数据集 test.data 示意图

启动 IntelliJ IDEA，编辑如下的 Spark MLlib 应用程序：

```
import org.apache.spark.{SparkConf,SparkContext}
import org.apache.spark.mllib.recommendation.ALS
import org.apache.spark.mllib.recommendation.Rating
import org.apache.log4j.Logger
import org.apache.log4j.Level
object ALSTest extends App{
    Logger.getLogger("org.apache.spark").setLevel(Level.WARN)
    Logger.getLogger("org.eclipse.jetty.server").setLevel(Level.OFF)
//构建 SparkContext 类型对象，封装应用程序属性信息与环境信息。
    val conf = new SparkConf().setAppName("ALSTest").setMaster("local[2]")
    val sc =new SparkContext(conf)
//读取并创建训练数据。
    val data = sc.textFile("/home/dong/spark-1.4.0/data/mllib/als/test.data")
    val ratings = data.map(_.split(',') match {
                        case Array(user, item, rate) =>
                            Rating(user.toInt, item.toInt, rate.toDouble)
                    })
//训练模型。
    val numIterations = 20
    val model = ALS.train(ratings, 1, 20, 0.01)
//对推荐模型进行评分。
    val usersProducts = ratings.map{ case Rating(user, product, rate)
```

```
=> (user, product)}
    val predictions = model.predict(usersProducts).map{
       case Rating(user, product, rate) => ((user, product), rate)
    }
    val ratesAndPreds = ratings.map{
            case Rating(user, product, rate) => ((user, product), rate)
    }.join(predictions)
    val MSE = ratesAndPreds.map{
        case ((user, product), (r1, r2)) => math.pow((r1- r2), 2)
    }.reduce(_ + _)/ratesAndPreds.count
    println("Mean Squared Error = " + MSE)
}
```

运行结果：

IntelliJ IDEA 中运行结果如图 8.22 所示。

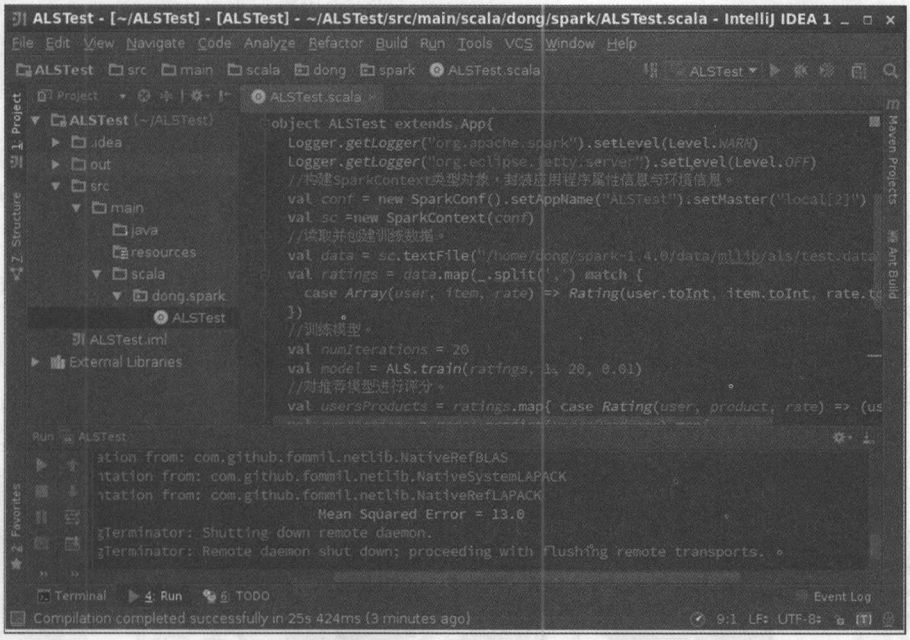


图 8.22 ALSTest 运行结果图

博文视点 · IT出版旗舰品牌

十载耕耘奠定专业地位

博文视点诚邀精锐作者加盟

以书为证彰显卓越品质

《C++Primer（中文版）（第5版）》、《淘宝技术这十年》、《代码大全》、《Windows内核情景分析》、《加密与解密》、《编程之美》、《VC++深入详解》、《SEO实战密码》、《PPT演义》……

“圣经”级图书光耀夺目，被无数读者朋友奉为案头手册传世经典。

潘爱民、毛德操、张亚勤、张宏江、昝辉Zac、李刚、曹江华……

“明星”级作者济济一堂，他们的名字熠熠生辉，与IT业的蓬勃发展紧密相连。

十年的开拓、探索和励精图治，成就**博**古通今、**文**圆质方、**视**角独特、**点**石成金之计算机图书的风向标杆：博文视点。

“凤翱翔于千仞兮，非梧不栖”，博文视点欢迎更多才华横溢、锐意创新的作者朋友加盟，与大师并列于IT专业出版之巅。

英雄帖

江湖风云起，代有才人出。
IT界群雄并起，逐鹿中原。
博文视点诚邀天下技术英豪加入，
指点江山，激扬文字
传播信息技术，分享IT心得

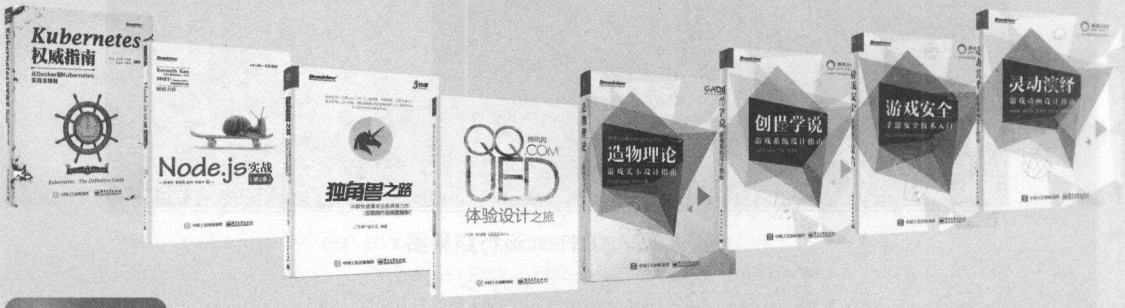
• 专业的作者服务 •

博文视点自成立以来一直专注于IT专业技术图书的出版，拥有丰富的与技术图书作者合作的经验，并参照IT技术图书的特点，打造了一支高效运转、富有服务意识的编辑出版团队。我们始终坚持：

善待作者——我们会把出版流程整理得清晰简明，为作者提供优厚的稿酬服务，解除作者的顾虑，安心写作，展现出最好的作品。

尊重作者——我们尊重每一位作者的技术实力和生活习惯，并会参照作者实际的工作、生活节奏，量身制定写作计划，确保合作顺利进行。

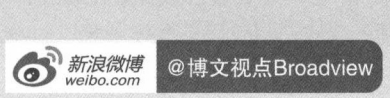
提升作者——我们打造精品图书，更要打造知名作者。博文视点致力于通过图书提升作者的个人品牌和技术影响力，为作者的事业开拓带来更多的机会。

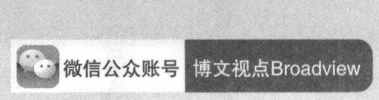


联系我们

博文视点官网：http://www.broadview.com.cn　　CSDN官方博客：http://blog.csdn.net/broadview2006/

投稿电话：010-51260888　88254368　　投稿邮箱：jsj@phei.com.cn





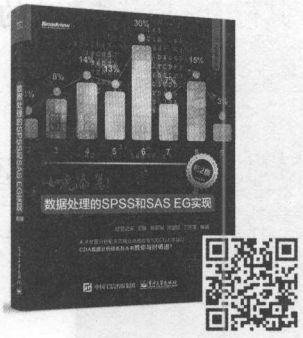
电子工业出版社 博文视点

CDA数据分析师系列丛书

CDA 数据分析师 CERTIFIED DATA ANALYST

经管之家主编
首套写给专业数据分析师的丛书



























大数据玩家（dataplay）：主要分享数据思维、数据分析、数据挖掘、数据应用、数据可视化等最新图书信息和重磅干货。

CDA数据分析师
CERTIFIED DATA ANALYST

CDA(Certified Data Analyst)，亦称“CDA数据分析师”，指在互联网、零售、金融、电信、医学、旅游等行业专门从事数据的采集、清洗、处理、分析并能制作业务报告、提供决策的新型数据分析人才。CDA秉承着总结凝练最先进的商业数据分析实践为使命，明晰各类数据分析从业者的知识体系为职责，旨在加强全球范围内正规化、科学化、专业化的大数据及数据分析人才队伍建设，进一步提升数据分析师的职业素养与能力水平，促进数据分析行业的高质量持续快速发展。

“CDA数据分析师认证”是一套专业化，科学化，国际化，系统化的人才考核标准，分为CDA LEVEL Ⅰ，LEVEL Ⅱ，LEVEL Ⅲ，涉及金融、电商、医疗、互联网、电信等行业大数据及数据分析从业者所需要具备的技能，符合当今全球大数据及数据分析技术潮流，为各界企业、机构提供数据分析人才参照标准。经管之家为中国区CDA数据分析师认证考试唯一主办机构，于每年6月与12月底在全国范围举办线下数据分析师考试，通过考试者可获得CDA数据分析师认证证书。

“CDA数据分析师培训”是根据CDA数据分析师认证体系标准而设立的一套专业化、科学化、系统化的学习方案。培训内容不仅包含认证标准中的技能知识要求，还有着企业环境中的真实项目和案例，能满足不同层次的学员需求，使学员能学到真本事技能并能够落地运用，实现商业价值。

品读经典，分享精华
我们期待您的加入

投稿邮箱：
zhanghm@phei.com.cn

交流学习：

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

Spark大数据分析
技术与实战

本书适合谁：

- 大数据技术初学者阅读；
- 作为高等院校计算机相关专业的研究生学习参考资料；
- 所有愿意对大数据技术有所了解并想要将大数据技术应用于本职工作的读者阅读。

- 问题讨论：本书读者可以在经管之家的“数据挖掘与商业智能（http://bbs.pinggu.org/forum-133-1.html）”板块中就书中的问题进行提问，也欢迎大家就自己遇到的业务问题和大家讨论。

博文视点Broadview
新浪微博 weibo.com
@博文视点Broadview

策划编辑：张慧敏
责任编辑：徐津平
封面设计：李　玲

上架建议：计算机>数据分析

ISBN 978-7-121-31903-7
9 787121 319037 >
定价：59.00元